The Deevolution of Concurrent Logic Programming Languages

Evan Tick
University of Oregon

CIS-TR-94-07
March 1994

Abstract

This article surveys the field of implementation of concurrent logic programming languages. I briefly review language semantics and programming paradigms, before summarizing the results of the past decade in compiler and runtime system implementation. A theme throughout the research presented is the *deevolution* of concurrent logic programming languages due to the limitations of what systems designers and compiler writers can efficiently implement, as well as the growing perception among programmers that reduced expressivity is sufficient.

This paper will appear in the *Journal of Logic Programming*, Tenth Anniversary Special Issue, 1995.

Contents

1 Introduction

There are two main views of concurrent logic programming and its development over the past several years. Most logic programming literature views concurrent logic programming languages as a derivative or variant of logic programs, i.e., the main difference being the extensive use of "don't care" nondeterminism rather than "don't know" (backtracking) nondeterminism. Hence the name *committed choice* or CC languages. A second view is that concurrent logic programs are concurrent, reactive programs, not unlike other "traditional" concurrent languages such as 'C' with explicit message passing, in the sense that procedures are processes that communicate over data streams to incrementally produce answers. A cynic might say that the former view has more academic richness, whereas the latter view has more practical public relations value.

This article is a survey of implementation techniques of concurrent logic programming languages, and thus full disclosure of both of these views is not particularly relevant. Instead, a quick overview of basic language semantics, and how they relate to fundamental programming paradigms in a variety of languages within the family, will suffice. No attempt will be made to cover the many feasible programming paradigms, nor semantical nuances, nor the family history. Excellent sources for this information are [22, 33, 34, 56, 72, 77].

The main point I wish to make in this article is that concurrent logic programming languages have been deevolving since their inception, about ten years ago, because of the following *tatonnement*:

- Systems designers and compiler writers could *supply* only certain limited features in robust, efficient implementations. This drove the market to accept these restricted languages as, in some informal sense, *de facto* standards.
- Programmers became aware that certain, more expressive, language features were not critically important to getting applications written, and did not *demand* their inclusion.

Thus my stance in this article will be a third view: how the initially rich languages gradually lost their "teeth," and became weaker, but more practically implementable, and achieved faster performance.

The deevolutionary history begins with Concurrent Prolog (deep guards, atomic unification, read-only annotated variables for synchronization), and after a series of reductions (for example: GHC (input-matching synchronization), Parlog (safe), FCP (flat), Fleng (no guards), Janus (restricted communication), Strand (assignment rather than output unification)), and ends for now with PCN (flat guards, non-atomic assignment, input-matching synchronization, and explicitly-defined mutable variables). This and other terminology will be defined as the article proceeds.

This view may displease some readers because it presupposes that performance is the main driving force of the language market, and furthermore that the main "added value" of concurrent logic programs over logic programs is the ability to naturally exploit parallelism to gain speed. Certainly the reactive nature of the languages also adds value, e.g., in building complex object-oriented applications. Thus one can argue that the deevolution witnessed is a bad thing when reactive capabilities are being traded for speed.

All is not lost in this regard: originating around 1988, a related family of languages and implementations has been developing around the Andorra model of computation, specifically Andorra I [16, 17], Andorra Kernal Language (AKL) [47], Pandora [7, 8], and ANDOR-II [83]. A key goal of the research is to combine "don't know" and "don't care" nondeterminism into powerful reactive programming languages. I cannot do justice to these rapidly *evolving* systems in this paper, because of space limitations.

This article is proceeds as follows. Sections 2 and 3 review the semantics of the simplest variant of CC languages, and present sample programs in a cross section of the language family to see the deevolution first hand. Section 4 discusses the entire plethora of implementation issues and related empirical research data. The article is summarized in Section 5.

2 Language Semantics

A committed-choice logic program[1] is a set of guarded Horn clauses of the form: "H :- $A_1, \ldots, A_m : T_1 \ldots, T_n \mid B_1, \ldots, B_p$" where $m, n, p \geq 0$. H is the clause head, A_i is an Ask guard goal, T_j is a Tell guard goal, and B_k is a body goal. In general, goals are user-defined and builtin procedure calls. However, in flat languages, guards are restricted to builtins. Ask guards passively match incoming arguments, whereas Tell guards can create bindings via unification. The ':' operator separates the guard types, and the commit operator '|' divides the clause between the guards and body. If $p = 0$ the clause is called a *unit clause*. A procedure is comprised of a set of clauses with the same principle functor and arity for H.

Informally, a procedure invocation *commits* to a clause by matching the head arguments (passive unification) and satisfying the guard goals. When a goal can commit to more than one clause in a procedure, it commits to one of them nondeterministically (the others 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. Alternatively, the program *deadlocks* if only suspended goals remain. A third result is program *failure*, which is defined more formally below.

The following operational semantics is a minor variation of the standard transition system semantics for flat concurrent logic programs and is derived from Shapiro [77]. "Flat" language variants restrict guards to be builtins, which simplifies our discussion of semantics. In later sections, the implementation issues (but not formal semantics) of non-flat (deep) guards are addressed.

A computation state is a tuple $\langle G; \theta \rangle$ consisting of a goal G (a sequence of atoms) and a current substitution θ. The initial state $\langle G; \varepsilon \rangle$ consists of the initial goal G and the empty substitution ε. A computation of a goal G with respect to a program P is a finite or infinite

[1]The knowledgeable reader may wish to skip to the next section.

sequence of states $S_0, \ldots S_i, \ldots$ such that S_0 is the initial state and each $S_{i+1} \in t(S_i)$ where t is a transition function from S to $\mathcal{P}(S)$ (defined below).

A state S is a terminal state when no transition rule is applicable to it. The state $\langle true; \theta \rangle$ is a terminal state that denotes successful computation and $\langle fail; \theta \rangle$ denotes finitely failed computation. If no transition is applicable to a state $S = \langle A_1, \ldots, A_n; \theta \rangle$ $(n \geq 1)$ where $A_j \neq fail$, $1 \leq j \leq n$, then the state is deadlocked. The meaning of a program P is defined as the set of all computations of a goal G with respect to P. In the following, a renaming function (to rename the clause variables apart from the goal variables) is required, but is beyond the scope of this article.

Definition: Transition Rules

- $\langle A_1, \ldots, A_j, \ldots, A_n; \theta \rangle \xrightarrow{reduce} \langle (A_1, \ldots, A_{j-1}, A_{j+1}, B_1, \ldots, B_k)\theta'; \theta \circ \theta' \rangle$ if $\exists$ a clause C s.t. *rename(C)* = "H :- Ask : $Tell$ | $B_1, \ldots, B_k$" and $try(A_j, H, Ask, Tell) = \theta'$.
- $\langle A_1, \ldots, A_j, \ldots, A_n; \theta \rangle \xrightarrow{fail} \langle fail; \theta \rangle$ if for some j, and for all (renamed) clauses "H :- Ask : $Tell$ | $B_1, \ldots, B_k$", $try(A_j, H, Ask, Tell) = fail$.

□

Function *try* is defined in terms of *match* which tests if the selected atom from a goal matches the head of the selected clause without binding any of the goal variables.

Definition:
$match(A_j, H) =$

$fail$	if $mgu(A_j, H) = fail$
θ	if θ is the most general substitution s.t. $A_j = H\theta$
$suspend$	otherwise

□

Definition:
$try(A_j, H, Ask, Tell) =$

$\theta \circ \theta'$	if $match(A_j, H) = \theta \wedge test(Ask\theta) = success \wedge mgu(Tell\theta) = \theta'$
$fail$	if $match(A_j, H) = fail \vee (match(A_j, H) = \theta \wedge test(Ask\theta) = fail) \vee (match(A_j, H) = \theta \wedge test(Ask\theta) = success \wedge mgu(Tell\theta) = fail)$
$suspend$	otherwise

□

The definition of $test(Ask\theta)$, which is not important for our purposes, can be found in Shapiro [77]. Note that these semantics, for FCP(:), include *atomic* tell unification. In other words, the most general unifying substitution θ' of the tell guards is computed, and if successful, composed with the entry substitution θ. If not successful, the clause try fails or suspends; however, in these cases, no tell bindings are exported.

In the following sections, a weakened form of *eventual* tell guards is discussed. These tell guards are not involved in the clause try at all, but rather are evaluated in the body. The formal semantics change quite a bit (this is left as an exercise for the reader!). Essentially, eventual-tell languages have two types of failure: head matching failure and body unification failure. Either type of failure within a deep guard is not terminal in the sense that the parent clause try will fail, but the parent procedure invocation may still succeed (or suspend). However, either type of failure outside of a deep guard is terminal, i.e., the program fails.

3 Paradigms and Programs

In this section the CC language family and its deevolutionary history are illustrated by means of examples. Sample programs are presented for representative languages, proceeding from "most" evolved to "least" evolved. For example is meant to emphasize the added expressivity of the language and its benefits. This is a broad, albeit brief, introduction and the interested reader is encouraged to access the literature.

3.1 Concurrent Prolog

The following example of Concurrent Prolog is the dining philosophers problem [77]. The code below spawns a ring network of n philosopher tasks that communicate by nearest-neighbor shared `Fork` variables. A philosopher may receive an `eating/2` message from its neighbors (clauses 1 and 2), but can commit upon that message only if the second argument in the message has been bound to `done`. Otherwise, while its neighbor is eating, this philosopher suspends.

```
phil( Id, [ eating( _LeftId, done ) | Left ], Right ) :-
    phil( Id, Left, Right).

phil( Id, Left, [ eating( _RightId, done ) | Right ] ) :-
    phil( Id, Left, Right).

phil( Id, Left, Right ) :- true :
    Left = [ eating( Id, Done ) | NewLeft ],
    Right = [ eating( Id, Done ) | NewRight ] |
    eat( Done ),
    phil( Id, NewLeft, NewRight ).

?- phil( 1, Fork1, Fork2 ),
   phil( 2, Fork2, Fork3 ),
   ...
   phil( n, Forkn, Fork1 ).
```

The key point is the atomic tell unification in the third clause. In this instance, the philosopher attempts to send its own `eating/2` message on its `Fork` streams. Although these streams are duplex (read and written by neighboring tasks), the write attempt will fail if another task is already eating because the identifier `Id` will not match. If *both* neighbors are idle, then the tell unification to the duplex stream succeeds *atomically*. That means no race can occur for the second fork, and hence deadlock is avoided. This algorithm cannot be elegantly implemented without atomic tell unification. Other programming examples illustrating the added expressivity of Concurrent Prolog can be found in Shapiro [77] and Bougé [10].

3.2 Parlog

A bounded buffer in logic programming is represented as a difference list X-Y where the head of X is the next item in the buffer and Y is the tail of the buffer. Consider a bounded buffer as represented in Parlog:

```
mode receive(^,?,^)
receive( X, [ M | Ms ]-Tail, Ms-NewTail ) :- nonvar( M ) |
   M = X,
   Tail = [ _Slot | NewTail ].

mode send(?,?,^)
send( X, [ M | Ms ]-Tail, Ms-Tail ) :-
   M = X.

mode init_buffer(^)
init_buffer( [ _, _, ..., _ | Tail ]-Tail ).
```

Procedure receive/3 accepts a buffer (second argument), reads the head M of the buffer into output X, and writes a new unbound slot into the tail of the buffer. The new buffer is returned (third argument). The mode declaration, "mode receive(^,?,^)" states that the first and third arguments are written and the second argument is read by the invocation. It does *not* refer to the modes of any subterms within the arguments stated, e.g., it does not declare the mode of M (although the nonvar guard implies that M is input).

Procedure send/3 accepts a message and a buffer, and writes the message into the head of the buffer, returning a new buffer. Sending will suspend if the second argument D-list is empty, i.e., Tail-Tail. This means the buffer is *full*. Receiving will suspend if the head of the buffer M is unbound, meaning the buffer is *empty*.

An interesting application is a buffered merge, switching two streams into one (clauses for termination are not included):

```
mode merge( ?, ?, ? )
merge( In1, In2, Out ) :-
   receive( M, In1, NewIn1 ),
   send( M, Out, NewOut ) |
   merge( NewIn1, In2, NewOut ).

merge( In1, In2, Out ) :-
   receive( M, In2, NewIn2 ),
   send( M, Out, NewOut ) |
   merge( In1, NewIn2, NewOut ).
```

The critical point is the use of deep guards to conditionally receive a message from either input stream and write it to an output stream. Since all ports are buffered, message output may suspend even if message input succeeds. Furthermore, inability to read from one input buffer will attempt to read from the other input buffer.

3.3 Flat Guarded Horn Clauses

A simplified form of the classic bounded buffer example is shown below in Flat Guarded Horn Clauses (FGHC). The programming paradigm is a toy version of a process network with two tasks: a consumer and producer. The tasks are reactive in the sense that rather than computing a value, they are perpetually rescheduled as dictated by dataflow constraints. The consumer suspends until an instantiated `Car` arrives, and the producer suspends until an unbound slot appears in the buffer.

The program has been purposely written so that the consumer requires both the buffer and its tail, as separate arguments, whereas the producer requires only the buffer. Of critical interest is the call to `init_buffer/1` which is passed an instantiated D-list `Buffer-Tail`. This is permitted because FGHC supports (eventual) tell unification, cf. simple assignment (as in Strand, coming next).

```
init_buffer( Buffer ) :-
   Buffer = [ _, _, ..., _ | Tail ]-Tail.

producer( [ Car | Cars ] ) :-
   Car = ferrari,
   producer( Cars ).

consumer( [ Car | Cars ], Tail ) :- nonvar( Car ) |
   ride( Car ),
   Tail = [ _NewSlot | NewTail ],
   consumer( Cars, NewTail ).

?- init_buffer( Buffer-Tail ),
   consumer( Buffer, Tail ),
   producer( Buffer ).
```

There is some preliminary work aiming at formalizing the difference in expressive power between atomic and eventual tell unification [23].

3.4 Strand

The Strand version of the previous bounded buffer code requires the following changes. The key point is that full tell unification is disallowed: assignment (`:=/2`) only is supported. Thus a new predicate `decompose_buffer/3` is needed to split the D-list into its components.

```
decompose_buffer( Buffer-Tail, B, T ) :-
   B := Buffer, T := Tail.

?- init_buffer( Buf ),
   decompose_buffer( Buf, Buffer, Tail ),
   consumer( Buffer, Tail ),
   producer( Buffer ).
```

Strand is similar to *moded FGHC* [96] which restricts a variable to have a single producer. At runtime Strand enforces the requirement that an assignment's LHS be initially unbound. Moded FGHC correspondingly requires that corresponding LHS and RHS variables in tell unifications have opposite modes, and requires this verified at compile time. Furthermore moded FGHC restricts a given argument position in a procedure to be consistently moded in all clauses comprising that procedure, as does Janus, discussed next.

3.5 Janus

The bounded buffer example, in a slightly different form, is formulated below in Janus [71]. In this version, `cash` is exchanged for a `ferrari`. The syntax is different from the previous languages, but is essentially a disguised form of Horn clauses.

The critical point to note is that a logical variable X is annotated as a "teller" $!X$ or an "asker" X. A teller can make bindings whereas an asker can only read bindings. A variable is restricted to two occurrences, enforcing single-producer single-consumer streams. This facilitates implementations that perform local reuse of memory. For example, in the code below, the producer can reuse the list cell containing the `cash` for the `ferrari`.

```
producer( !Bs, Ds ) ::
   Ds = [ cash | Ds1 ] ->
      Bs = [ ferrari | Bs1 ],
      producer( !Bs1, Ds1 ).
   Ds = [] -> Bs = [].

consumer( Bs, !Ds ) ::
   Bs = [ ferrari | Bs1 ] ->
      Ds = [ cash | Ds1 ],
      consumer( Bs1, !Ds1 ).
   Bs = [] -> Ds = [].

?- producer( !Bs [ cash, cash, ..., cash | Ds ] ),
   consumer( Bs, !Ds ).
```

3.6 Program Composition Notation

A non-buffered producer-consumer example is shown below in Program Composition Notation (PCN) [11]. The syntax is C-like, with two critical distinctions. First, there are both logical (called *definitional*) variables as well as *mutual* variables. Second, control blocks are annotated as either sequential (';') or parallel ('||'). There are three rules supported by the language implementation that guarantee correct management of the two types of variables:

§1. A mutable variable can be shared by blocks in a parallel composition only if no block modifies the variable.

§2. When a mutable variable occurs on the RHS of a definition statement, the current value of that mutable variable is copied and the definition then proceeds if a definitional variable were involved.

§3. When a definitional variable occurs on the RHS of an assignment, the assignment suspends until the variable has a value and then proceeds.

In the example below, §2 is invoked at statement (1), allowing `state` to be mutated in statement (2). However, `state` in statement (3) is definitional.

```
producer( S )
double state[ SIZE ];
   produce( S, state )
produce( S, state )
double state[];
   {; S = [ msg( state ) | Ss ],           (1)
      update( state ),                     (2)
      produce( Ss, state )
   }

consumer( S )
S ?= [ msg( state ) | Ss ] ->              (3)
   {|| use( state ),
       consumer( Ss )
   }

goal()
   {|| consumer( S ),
       producer( S )
   }
```

What a long, strange trip it has been! Atomic tell unification (and read-only variable synchronization) in CP was weakened into eventual tell unification and input matching synchronization in Parlog. Deep guards in Parlog were weakened into flat guards in FGHC. Body unification in FGHC was weakened into assignment in Strand. Multiply shared variables in Strand were weakened into single-producer single-consumer (single occurance) variables in Janus, and declarative/mutable variables in PCN. In the remainder of the article, I will discuss how these languages have been implementated.

4 Implementation Issues

Efficient implementation of CC languages, as that of more traditional languages (such as explicitly message-passing imperative languages) hinges on low memory usage, compile-time code optimization, and low-overhead runtime management of concurrency. By keeping the program's working set small, locality of the machine's memory hierarchy can be best exploited, reducing expensive faults farther from the CPU. The storage model selected, e.g., stack or heap, is a critical design decision here. Compile-time code optimization also centers around the memory hierarchy: efficient utilization of the available machine registers and cache. This involves avoiding redundant computation (e.g., by strength reduction of loops afforded by dataflow analysis [1]), which also saves CPU cycles. Finally, the runtime overheads of concurrent task management must be significantly lower than computation within tasks. This is often noted in terms of the communication to computation ratio, assuming that the primary action of task management is transmission of messages between processors. However, note that the cost of task creation, switching, and scheduling, are very important as well.

Implementations to date of CC languages were targeted to the broad categories of uniprocessor, shared-memory multiprocessor, and distributed-memory multiprocessor hosts. Almost all of the implementations use storage models wherein tasks are allocated in an *ad hoc* fashion from either global or local storage pools, i.e., procedure invocations are packaged as individual tasks facilitating concurrent suspension and resumption. A stack-based storage model, wherein a task is composed of procedure invocations executing on a stack, is more efficient if average task lifetimes are sufficiently long. However, task suspension must still be implemented, perhaps in a manner similar to implementations of the *freeze* primitive in certain Prologs [58]. In the long-run, although not seen yet, implementations for multiprocessor hosts must also move to stack-based models, perhaps adopting ideas from partitioning of threads in dataflow languages [21, 92]. In this section I review the main themes and efforts in the implementation of CC languages over the past decade. Let us begin with a brief historical overview.

4.1 History

The past ten years witnessed an explosion in the research productivity in developing parallel logic programming systems. The specific subfield of concurrent logic programming system development was quite active with primary research groups at the Weizmann Institute of Science, the Imperial College of Science and Technology, and the Institute of New Generation Computer Technology (ICOT). One milestone was 1982, the first year of ICOT's operation, when E. Shapiro during a visit designed *Concurrent Prolog (CP)*, the seminal committed-choice language [76]. This work was influenced by the Relational Language by K. Clark and S. Gregory [15], which had elements of committed-choice languages. But it was Shapiro's much-cited ICOT TR-003, published in winter 1983, that formed the blueprint for much of the language and operating system design work that followed, similar in impact to D. H. D. Warren's abstract machine (WAM) definition [98], published in summer 1983. The history and influences of the family of languages is described best in Shapiro [77]. Language evolution was so riotous that system implementation could hardly keep up.

Some interesting comparative work done at the University of Edinburgh by R. Trehan [93] and H. Pinto [70] summarized the experiences of programming and interpreting these languages in the "early days." These concurrent languages could be differentiated primarily by their synchronization mechanisms and how they managed multiple local environments. There are various other attributes, such as granularity control and goal scheduling, unification, etc., that affect implementation complexity and efficiency. These are discussed in depth in the following sections.

Trehan and Pinto's studies focused on interpretation, whereas further evolution of implementation efforts led to compilation and hardware support. The first abstract machine designs for this family of languages were the WAM-like Flat Concurrent Prolog (FCP)[2] machine (Emu) by A. Houri [43, 80], Sequential Parlog machine (SPM) by S. Gregory *et al.* [36, 38], and the KL1 machine by Y. Kimura [48]. These systems represent the first-generation compiled implementations of concurrent logic languages, evolving into more sophisticated systems. The sequential FCP machine was refined first by S. Taylor into a distributed-memory multiprocessor implementation on a hypercube [85] and by S. Kliger into a RISC-based abstract machine and optimizing compiler [51]. The SPM led to J. Crammond's Abstract Machine (JAM), the first parallel Parlog implementation [20], and the PPM [12]. The KL1 machine was implemented on shared-memory machines as Panda

[2]There are several variants of FCP as defined by Shapiro [77]. In this paper I leave the precise variant unspecified unless relevant.

[74] and evolved into the abstract machine shared among the "parallel inference machines" (PIMs). A hybridization of a few of these projects was I. Foster and S. Taylor's flat Parlog machine [28] leading to the Strand Abstract Machine (SAM) [29], ported to several types of multiprocessors. these systems represent the second-generation parallel implementations, the first comparative study of which was by Foster and Taylor [28]. The community is now completing the construction of third-generation *optimized* compiler-based, portable systems, e.g., the jc Janus system [39], Monaco [88], and a portable KL1 system [13].[3]

Specialized hardware efforts were concentrated mainly at ICOT with the decade-long FGCS project and their aim of building PIMs. The personal inference machines (PSI-I,II,III) [84, 63] were followed by mockup PIMs (Multi-PSI-V1 built of 6 PSI-Is, and Multi-PSI-V2 built of 64 PSI-IIs), and finally PIM/{c,i,k,m,p} [61, 75, 6, 64, 54]. The main efforts, PIM/{m,p}, are large multiprocessors (2^8–2^9 processors) based on specialized hardware for "direct" execution of KL1 (either by microcode or RISC-based intermediate machine languages). Other notable hardware implementation efforts include the Carmel microprocessors [41] and a related microprocessor proposed by Alkalaj [3]. A full analysis of hardware issues in concurrent logic language implementations is beyond the scope of this article (see Tick [87] for instance), although I do correlate the instruction set designs of the software and hardware oriented implementations in Section 4.9.

In summary, the seminal research results in CC language implementations are:

- Shapiro [76] and Mierowsky [59]: first interpreters.
- Emu [43, 80], SPM [36, 38], and KL1-B [48]: first abstract machines.
- Taylor [85]: first distributed implementation.
- Strand [26, 29]: first robust, high-performance, scalable, compiler-based implementation.
- JAM [18] and Panda [74]: first implementations optimized for shared-memory multiprocessors.
- PIMs [6, 54, 61, 64, 75]: first custom hardware implementations.

4.2 Principles and Trends

Efficient implementation of concurrent logic programs requires strong foundations in several areas. As in any parallel system, task[4] switching and task creation are the primitive operations that must be made fast. Furthermore, as in any computational system, task invocation, variable binding, and memory reclamation must also be made fast. For concurrent logic programs, task switching means suspending one task and substituting (resuming) another; task creation means building a body goal task from its parent's arguments and perhaps spawning it on a remote processor. Task invocation is extended here to include the

[3]Interesting comparisons of the execution performance of many of these first, second and third generation systems can be found in Taylor [85] and Tick [88, 89].

[4]The words *task*, *process*, and *goal* are used interchangeably in this article.

action of executing a goal to the point when it commits, i.e., performing the clause tries needed to commit, suspend or fail.[5] Variable binding incurs added overheads to guarantee atomicity (i.e., locking around the update to avoid races among competing writers). Not only is fast memory reclamation critical, but moreover so is efficient use of memory in the first place, since the single-assignment nature of the languages can be quite profligate in touching memory.

By far the most complex implementation aspect of these basic operations is task switching and task invocation because of language synchronization semantics that require implicit synchronization on potentially incoming procedure arguments. This places a burden on the compiler and generally bloats procedure invocations with respect to sequential languages and implementations. The various nuances of language semantics, e.g., deep or flat guards, atomic or nonatomic tell unification, impact implementation efficiency.

Orthogonal to these primitive operations are intelligent task management policies that are desirable: balanced load, balanced granularity, and fair scheduling. These concepts are not unique to concurrent logic programs, and are required independent of how fast the primitive operations can be made. Looking at underlying multiprocessor hosts, an additional requirement exists to achieve full efficiency: latencies must be hidden. Memory latency in distributed multiprocessors is the major problem to dealt with. As Arvind showed [4], hiding latency effectively is directly traded-off against switching tasks quickly. We shall see (Section 4.5) that current concurrent logic programming systems can hide latency, but only within limits, and certainly overly-complex languages features cannot be effectively hidden.

The past ten years have seen a trend towards *deevolution* of logic programming languages driven by the practical need to build fast implementations. The most drastic step was the definition of committed-choice languages that did not backtrack, enabling the first pseudo-parallel interpreters to be built. The next deevolutionary step was from deep to flat guards, and moving from synchronizing on dynamic read-only variables to synchronizing on statically-declared arguments, enabling the first efficient implementations to be built. Next were restrictions placed on how variables could be bound: Strand [29] abolished output unification in favor of assignment, similar to moded FGHC [96] which constrains a logical variable to have a single producer.[6] More strict, Doc [42], $\mathcal{A}'\mathcal{UM}$ [100], and Janus [71] constrain a logical variable to have at most a single producer and single consumer. These simplifications facilitate compile-time analysis and optimization of memory usage.

The progressions are further described in subsequent sections. The key point is that languages are refined by reaching an equilibrium between what application writers *demand* and what implementors *supply*. There is not yet full agreement as to where this equilibrium point is for concurrent logic programs, and I think it will be most strongly influenced by fast, portable, and parallel implementations.

[5]A concurrent logic program task is like a *thread* in threaded architectures. The task invocation creates a main thread which may split into multiple threads during guard evaluation, all synchronizing at commit, leading to a single clause body thread. The body calls spawn new threads, and so on.

[6]The two languages are *dissimilar* in that Strand checks at runtime if the LHS of an assignment is a variable, while this is guaranteed at compile time in moded FGHC.

4.3 Synchronization

Concurrent logic programs synchronize on logical variables, similar to how non-strict data-flow languages use I-structures [5, 55]. For a given clause, a required input variable (also called a *synchronizing* variable) is informally a variable for which a value is necessary to test matching in the head or guard. If a required input variable is unbound upon procedure invocation, the corresponding clause cannot commit. Furthermore, if no clause defining the procedure can commit and not all clauses fail, it implies some required input value(s) have not been delivered, and the task must be suspended. Concurrently, if any of these required input variables are bound, the task must be resumed.[7]

Input matching (synonymous with passive unification) is transformed by compilation into instruction sequences that make matching efficient in general. The ability to synchronize on variables requires temporarily binding certain unbound logical variables to a suspended task to enable subsequent resumption. The efficiency of this infrastructure is the main factor in synchronization performance.[8]

The FCP, JAM and KL1 machine architectures all use similar methods of "hooked" variables, i.e., assigning indirect pointers from suspended variables to process structures [43]. Indirection is required to allow both multiple variables to synchronize the same task, and multiple tasks to be synchronized by the same variable. Unbound variables are infrequent data types: Imai and Tick [45] measured 1–15% of dynamic objects are unbound variables across a KL1 benchmark suite. To our knowledge, no one has measured the prevalence of hooked variables, and the characteristics of those hooks. It is a widely-held belief that hooks are quite simple in structure and rare in frequency.[9] Thus JAM Parlog [20] and Strand [29] allow goals to be hooked to only one variable, thereby obviating the complex bookkeeping structures needed for the general case. JAM exploits shared memory to implement a "hybrid" suspension list to gain this efficiency. Singly-suspended tasks are simply linked together in a daisy chain emanating from the unbound variable (since resumption will disperse the entire chain). Multiply-suspended tasks are "wired" into the chain via *suspension notes* and *hangers*, the standard indirection mechanisms [43, 20] needed to guarantee that bindings to alternatively suspended variables do not chase dangling pointers. Strand initiates all suspensions as if they are the single-variable type, and if this most frequent case is violated, the suspended task is added to an exceptional (global) queue. This queue is accessed only if all processors become idle. No measurements have been presented indicating the utility of this method.

An orthogonal issue is how to specify the input variables upon which to synchronize. The most common method is "procedure level" representation wherein synchronizing vari-

[7] *Resumption* is defined here as reattempting to execute the task, and therefore binding the variable is sufficient to resume the task. However, the binding is not necessarily sufficient to permit the task to commit: the task may suspend again.

[8] Early systems did not attempt to statically analyze logical variables, e.g., to determine if a variable can possibly be hooked, and if not, how to generate more efficient code for the ask tests. Recent compilers, e.g., [51, 99], claim to do global static analysis to determine this and other information.

[9] The former assertion is more strongly supported than the latter — "object-oriented" programs can create many suspensions, as discussed in the remainder of this section. The two such programs measured by Imai and Tick produced far more variables than the other benchmarks.

ables are syntactically specified (explicitly as in Parlog or implicitly as in GHC) on a per clause basis. Alternatively synchronization at a "data level" representation specifies synchronizing variables, e.g., "read-only" variables in Concurrent Prolog. The latter method has gone out of favor because, although it facilitates certain sophisticated systems programming techniques, it complicates dereferencing and unification, and frustrates static analysis. The elegant programming techniques it enables are rarely used in applications programming [77], yet the cost of implementation is felt throughout the design [28], primarily because it requires atomic unification support. Foster and Taylor [28] measured (for small benchmarks executing on a sequential workstation) that trailing needed to support atomic unification (discussed further in Section 4.6) in FCP(:) (with atomic unification [77]) caused a 5% degradation in performance compared to flat Parlog (without atomic unification). For programs with suspension ratios (# suspensions/# reductions) of 19–56%, additional degradation of 4–8% was observed, hypothesized as other overheads associated with read-only variables (since flat Parlog and FCP(:) were calibrated except for that).

Another implementation issue is the actual control flow of checking the synchronizing variables (discussed at length in Section 4.4). It was originally believed that parallel execution of the clause tries was beneficial because it implied faster invocation. However, if deep guards are permitted, then parallel clause tries require the ability to sustain multiple environments and incur most of the problems associated with OR-parallel management of bindings under search for a single solution. Furthermore, with flat guards, the little amount of work within the clause tries may not justify the overhead of executing them in parallel. Crammond showed consistently negative speedups (3.8% to −32%) for small, flat-guarded benchmarks on JAM Parlog (executing parallel clause tries) on a shared-memory multiprocessor [18]. In fact, compilation techniques such as decision graphs [51] remove redundant computations among the clause tries, furthering the argument that parallel tries do not pay for themselves.

Sato and Goto [74] showed, for the shared-memory Panda system, that suspension induces execution overhead of 1–5% for small benchmarks, because of the necessity to redo the clause tries on resumption. Especially with decision-graph compilation techniques, it is not easy to avoid recomputation since there is more sharing among the code generated. For Panda benchmarks with low suspension ratios of 1–8%, depth-first scheduling mechanism effectively suppressed suspensions (with respect to breadth-first scheduling), but the benchmarks were quite simple. The one Panda benchmark with a high suspension ratio of 42% was not suppressed by depth-first scheduling. Taylor [85] measured suspension ratios of 0–56% on the hypercube for small programs, including an assembler. The higher ratios are due to static pragma-driven scheduling on the hypercube, compared to the dynamic scheduling on Panda. Imai and Tick [45] measured 14 medium-sized benchmarks ranging from 0.3–67% suspension ratios, with a geometric mean of 3.7%.

All these statistics taken together indicate, among other things, that suspensions are not infrequent and thus overheads associated with suspensions can seriously degrade execution performance. The problem is total lack of knowing in what order or *schedule* the concurrent goals will execute. However, given certain information, for instance knowledge of dependencies among goals, suspensions can be effectively neutralized. Techniques to collect such information include abstract interpretation [50, 51] and constraint propagation [96, 90].

For example, Kliger [51] reports that a set of 27 small-to-medium size FCP(:) benchmarks achieved 21% geometric mean speedup due to a set of optimizations based on the global schedule analysis. Knowing a partial order of execution engendered optimizations including reduction of (atomic tell) unification into assignment, in-lining arithmetic, and efficiently manipulating unboxed objects (i.e., conducting a chain of arithmetic operations on data cells with tags masked out).

4.4 Guards and the Process Structure

Guards, similar in purpose to Dijkstra's *guarded commands*, were introduced to logic programming in the Relational Language [15]. They extend, from simple input matching, the expressivity of how to commit to a clause. In their most general form, guards among clauses defining the same procedure represent disjunctive processes racing to commit. Implementation difficulties occur 1) if these processes are allowed to bind (nonlocal) variables, and 2) even if binding is outlawed, if processes are permitted to make nested calls. The former problem is indicative of "unsafe" languages, and the latter problem is indicative of languages with "deep" guards. Considering the range of complex to simple implementations, the languages fall into three basic categories: unsafe and deep (e.g., Concurrent Prolog), safe and deep (e.g., Parlog), and safe and flat (e.g., FGHC).[10]

Unsafe clauses may compete with one another in the sense that each may wish to make conflicting bindings to the same (nonlocal) variables. This is implemented by restricting bindings to a local environment, for exportation upon commit. Exportation can, however, conflict with concurrent bindings made nonlocally. If this happens, the clause try fails. Detecting inconsistencies is a major implementation problem in these languages — there is a choice among detection *before* commit ("atomic") or *after* commit ("eventual"). The former presents a clearer semantic model to the programmer, but is far more difficult to implement (see Section 4.6). Programmers have a more difficult time debugging eventual-tell unification languages because such body unifications can be executed (and fail) some significant time after (due to scheduler delays) the parent procedure successfully committed.

Deep guards effectively form a process hierarchy or tree, with local environments at each level. Local environments are needed, even if the language is safe, because incoming bindings (to local variables) must be saved across deep guard evaluation. In other words, the arguments must be cached in a unique environment because the evaluation of deep guards may involve further procedure invocations (possibly recursive) with their own environments. One severe implementation problem is the management of multiple environments (one per deep-guard clause in the same procedure) if guards are evaluated concurrently. This *value access control problem* is similar to that of OR-parallel implementations of Prolog: how to efficiently ensure that only ancestor environments on a path to the root are accessible, and that all other environments are hidden.

Another implementation problem is supporting *fair* execution while descending the hierarchy, while retaining low complexity and cost. If fair execution is not guaranteed then

[10]A subtle issue is *how* the languages are made safe. Whereas in Parlog body goals executed in deep guard evaluation must not attempt to export observable bindings, in GHC body goals executed in deep guard evaluation must *suspend* when they attempt to export observable bindings.

eagerly executed guards may loop, preventing later guards from failing and freeing up the computation. Shapiro [77] states that the inability to achieve fairness at low cost motivated flat languages.[11] He cites early CP implementations (e.g., [60]) as either unfair or of "unacceptable" complexity.

JAM Parlog [20] constrains deep guards to be used only in clauses bracketed by *sequentialized clause separators* (in some languages called *otherwise* guards). Such separators prevent subsequent clauses from being tried until all previous clause tries fail. This restriction obviates concurrent evaluation of deep guards, simplifying management to that of a single local environment per procedure. Crammond [20] states that this restriction allows most of programmers' intended uses of deep guards, e.g., as if-then-else conditionals.

Parlog offers the programmer a sequentialization operator g & b that guarantees goal g executes to completion before goal b is executed. In JAM, the implementation views a clause as compiled above with the guard g and body b. In other words, the same mechanism used to implement sequential goal execution does double duty for deep guard execution. Deep guards need to be evaluated concurrently to avoid deadlock; however, given mode information, flat guards can often be executed in-line for efficiency. The environment necessary for carrying local bindings over a sequentialization operator are not unlike a Prolog environment in standard WAM implementations, cf. goal stacking in standard CC language implementations.

Restricting the language to only safe, flat guards engendered decision-graph compilation [51] because clause tries can be compiled in line without transfer of control nonlocally to other goals. A decision graph is composed of if-then-else and switch nodes which transfer local control conditionally upon a test. A graph is formed, rather than a tree, to guarantee space proportional to the number of clauses in the procedure. To ensure space linearity, a clause is propagated down one and only one branch of the graph as code is being generated. Thus clauses ambiguous to a test are conservatively placed in a *continuation* branch, and sibling branches jump to the continuation upon failure.[12] For a suite of 27 medium-size benchmarks, decision graphs executed 3.2 times faster on average than WAM-like compilation [51]. The code size expanded by 30% on average, with a particularly degenerate program (Salt & Mustard [86]) doubling in size. An interesting problem is how to order the graph nodes, and how to generate optimal code for the tests, conditional branches and switches, to minimize execution time [24].

A main purpose of deep guards is to perform speculative computations that can fail allowing alternative solutions to succeed. Unsafe languages enriched this paradigm allowing bindings to be made along the speculative path. Experience has shown that support of both of these operations is too expensive for the low frequency with which they are used. The deevolution to flat languages is complete in the sense that almost all research groups opted to reduce language expressibility in favor of easily-implementable flat guards. In a further

[11]This problem still exists, in a less troublesome form, for unfair flat languages when early builtin guards suspend, preventing or delaying later guard failure. This can only reduce the failure set.

[12]When generating a decision *tree*, testing a variable for which a group of clauses "don't care" requires copying those clauses to *each* branch of the test, thus failing to achieve space linearity. The space complexity with respect to the number and type of guards cannot be easily formalized because of the potential of nonmutually exclusive conditions.

extreme, Fleng [66] abolished traditional guards in an effort to streamline execution. In general, guard (ask) tests must be pulled up and evaluated at each call site. This allows the optimization wherein certain guard tests need only be evaluated at certain call sites. Global analysis is needed to produce the information required for this optimization. Although FCP has guards, a similar optimization is enabled by Kliger's method of customizing decision-graph clause tries for different call sites [51].[13]

4.5 Reading and Writing Logical Variables

The costs of reading and writing[14] logical variables can be calculated as the *frequency* of operations required, multiplied by the *cost* of the operations. For example, reading a logical variable incurs the incremental cost of suspending the variable at the rate of suspension. In shared-memory multiprocessors, all accesses are "local" (i.e., do not travel across a high-latency network) so that the relevant overheads are lock traffic on the shared bus and lock contention. Contention, i.e., multiple concurrent requests of the same lock, can be exacerbated when the host does not supply enough physical locks for all objects needing locks.[15] For small benchmarks, Sato and Goto [74] reported that locking accounted for only 1–5% performance degradation on the Sequent Balance. This conveys both the relative efficiency with which locks can be implemented with shared memory, as well as the retained significance of lock overhead. Interestingly, although most of the lock traffic they measured was for protecting bindings, most of the observed lock contention was for bookkeeping locks for scheduling and termination.

Distributed-memory multiprocessors are significantly more problematic because of the overheads incurred in reading and writing nonlocal variables. Nonlocal reading requires sending a message requesting the variable's value, and receiving a reply. Nonlocal writing requires issuing the binding — the receiver can update the variable locally (without explicit locking) and send either a success or failure acknowledgement. The incremental cost of resuming tasks hooked to the bound variable must be accounted for in a macro view of execution.

There are of course variations on both of these protocols. Taylor [85] discusses a protocol on a hypercube where nonlocal writes first request a remote lock, and upon receiving the lock, issue a remote write. He measured 61%–100% of all messages sent, for six FCP benchmarks, are nonlocal reads. The four smallest benchmarks required an arithmetic average of 99% reads. Although write frequency is seen to be very low, its amplified cost can be felt. For example, Taylor demonstrated that for the incomplete message paradigm (where

[13]Practical definitions of Fleng allow the programmer to specify guards, which are then pulled up to the call sites. Without global analysis, of the complexity required by Kliger, *all* guard tests must be pulled up to *each* site.

[14]Throughout this section "read" and "write" refer to *logical* not *physical* operations. For instance "reading" a variable may actually involve hooking a goal on that variable, which would involve a physical write (store) operation.

[15]For example, the Sequent Balance and SGI MIPS-based multiprocessors offer a limited number of locks, whereas the Sequent Symmetry allows every memory location to be locked. Because this hardware attribute cannot be easily modified for a given host, studies of lock contention *vs.* lock granularity have not been performed for concurrent logic languages.

nonlocal reading and writing occur with equal frequency), the main execution overhead on a multiprocessor was not the sender reading the return value, but rather the receiver locking and writing the return value. In this simple example, locking proved to be extremely expensive (performance degradation of two times on two hypercube nodes) because the latency could not be hidden.

Reducing the cost of reading is critical in distributed implementations. If a complex term is to be read nonlocally, an important design consideration is how much of the term should be eagerly transferred. Taylor also examined the affect of this *copy depth* parameter on performance. For standard paradigms such as producer-consumer and incomplete messages, performance improved significantly for initial increases in copy depth, after which no improvement was seen.[16] The interpretation of these results is that the consumer is "brought up to speed" by increasing transfer size until the point at which it outruns the producer, after which no further improvement can be achieved. Because these are such pervasive programming techniques in concurrent logic programs, it is imperative to find ways to speed them up. Hardware support for message management (packing and unpacking, merging active messages straight into the execution pipeline) to more effectively hide latencies is one approach, similar to the goals of threaded architectures (e.g., Nikhil *et al.* [65]). Another idea is to reduce the number of messages sent, either by introducing new programming paradigms, or by dynamically migrating tasks and streams so that communication is local. Yoshida [100] took the latter approach in the design and implementation of $\mathcal{A}'\mathcal{UM}$, discussed in Section 4.10.

4.6 Unification

Unification is somewhat controversial because it stands out as one of the few unbounded-time operations required by logic programs compared to conventional languages. In many cases unification can be compiled into simple instructions, as was elegantly shown in the WAM [2, 98]. Unification in committed-choice languages can be categorized as either *input* (also: *passive* and *ask*) or *output* (also: *active* and *tell*), reflecting the exportation of bindings. Ask unifications implement head matching either as explicitly compiled match instructions or as invocations of a fully general passive unify routine. Luckily, full passive unification is rarely executed: it occurs only when checking the equality of two incoming arguments. For example, Foster and Taylor [28] measured the execution of 153,800 matching operations and 15,300 general passive unifies (9% of total) in an Assembler benchmark written in Flat Parlog. Furthermore, if sufficient type information is inferred, general passive unifies can be reduced to simpler tests.

At the leaves of unification's recursive descent, rules for unifying primitive data types come into play. Read-only synchronization requires an extended set of rules [60, 85] compared to procedure-level synchronization, potentially reducing performance.

Whereas ask unification occurs before commit, the location of tell unification varies among the languages. Unsafe languages require *atomic* tell unification wherein no output bindings are seen until commit. This means that bindings must be locally trailed, and per-

[16]The determination of the copy depth parameter may possibly be done at compile time with strictness analysis of data, e.g., Wadler [97].

haps undone upon failure. Furthermore, atomic unification can bind two variables which are inputs of the same procedure invocation. This raises the issue of whether such bindings should be eagerly acknowledged, although such an implementation causes a performance degradation of 1–29% [28]. This and trailing overheads led to the abandonment of atomic unification for implementation reasons alone. Safe languages place tell unifications after commit (called *body unification*). The implementations are thus free to perform body unifications on the fly, with unification failure causing the unification goal's parent procedure to fail.

Even body unification is complex when considering multiprocessor implementations. The main problem is to avoid potential race conditions among concurrently executing tasks. Thus any logical variable that needs to be bound must be locked first. Even if two variables are to be unified, *both* must be locked to prevent a competing task from creating a cyclical binding. Furthermore, they cannot be locked in an *arbitrary order* under threat of deadlock with the competing task trying to lock them in the reverse order (very unlikely, but possible, unless mode restrictions are known, as discussed below). Some ordering must be made, e.g., exploiting the nature of a shared-memory name space.

By making the most frequent case fast, general unification on multiprocessors can be implemented efficiently. The most common case by far is binding a non-variable to an unbound logical variable that does not require dereferencing. A fast stub can be constructed that tests if one operand is a non-variable, and one is unbound and not hooked. On a shared-memory machine, the variable is then locked, checked if still unbound (i.e., that some competing task didn't race to bind it), the binding is made, and the cell is unlocked. This sequence can be significantly sped up in a safe language on a multiprocessor with atomic exchange.

Otherwise, if the initial condition is not met, full unification is required: the two operands to be unified must be locked if unbound, dereferenced, and compared or bound. On distributed-memory multiprocessors, the same algorithms can be naively used, potentially with nonlocal accesses required for each simple unification.

Because of the implementation complexity and potential execution overheads of output unification on distributed-memory multiprocessors, and the evidence that it is infrequently used in its full generality, output unification further deevolved in Strand to *assignment*. Thus recursive descent is obviated, and the left-hand side of the assignment is required to be a variable. However, this does not rule out the need to test for exceptions, or hooked variables for which the associated task(s) need to be resumed. Thus binding is still expensive compared to imperative assignment.[17] Recall (Section 3.6) that PCN offers both *definition* (logical) and *mutable* variables. Thus safe and efficient assignment to mutable variables can be guaranteed by the programmer. Furthermore, memory usage can be reduced by destructive update of mutable variables. In a sense, PCN is the farthest deevolution has progressed in concurrent logic languages.

[17]With compile-time freeness analysis, e.g., [96], and hookedness analysis, e.g., [99], these tests can be safely removed.

4.7 Task Scheduling and Priority

There are various philosophies for automatic scheduling of parallel tasks. Compile-time analysis can be attempted to determine a fixed schedule mapping tasks to processors. Run-time profiling information can aid the static analysis. A radical departure is to perform all scheduling dynamically without any static aid, or a hybrid combination of static and dynamic. Another approach is to avoid automation and require the programmer to explicitly distribute tasks.

Automatic scheduling in concurrent logic programming systems is usually *dynamic* and *process-oriented* (e.g., JAM, Panda, Monaco) because tasks are too small, undifferentiated, and numerous to allow practical static analysis. For shared-memory multiprocessors, the main implementation issue is how to efficiently manage the goal queues. A single shared queue would eliminate the need for load balancing, but contention for this scarce resource is too costly. Splitting the queues up, one per processor, removes contention but leads to potential unbalancing. Once queues have been split, there emerges the implementation paradigm, on large-grain process systems such as UNIX, of *task farming.* Here a single UNIX process, often called a "worker," is responsible for coroutining between the execution and scheduling of goals. In *on-demand scheduling*, goals are not eagerly distributed among workers and only an idle worker searches for work, thereby minimally disturbing busy processors. Sato and Goto [74] and Crammond [19] examined variations of on-demand scheduling involving further splitting the local queue into *private* and *public* queues and allowing idle workers to steal only public work. Crammond reported that for eight medium-sized Parlog benchmarks, private/public queues offered slightly better and more consistent speedups than public-only scheduling, on the Symmetry and Butterfly II (on 16 Symmetry PEs, geometric mean efficiencies, i.e., speedup/16, of 86% and 83%, respectively). These early studies measured multiprocessors with far slower processing elements than are available today.

There has also been much work within the ICOT FGCS Project exploring automatic load balancing methods, e.g., [32, 44, 61, 74, 81]. The most successful experiment has been the *multi-level load balancing* (MLLB) scheme for balancing OR-parallel search programs on a distributed-memory multiprocessor [32]. The idea is to partition the available processors into groups, and allocate one distribution master per group. Slave processors within these groups request work from the master. The master receives work from a global master whose function is to distribute "super" work granules to the group masters. There is a method of merging groups, and given the regular nature of OR-parallel search, this method has been shown to be quite effective, e.g., speedup of 50 on 64 processor Multi-PSI for the pentomino benchmark [32].

The drawback of MLLB is its limited application domain. Thus even ICOT resorted to explicit user-defined "pragma" in the KL1 language for remote task scheduling on distributed multiprocessors. Strand and PCN also require pragma. In these latter languages, the user is encouraged to design load-distribution management networks, called *motifs*, e.g., MLLB could be specified as such [27, 30]. In PCN, motifs consist of several programming constructs implemented in the source language with libraries providing support. Simple pragma are enriched by allowing the definition of *virtual topologies*, which can be embedded within physical topologies. Topologies are collections of nodes, such as a hypercube network, implemented by process structures. User-defined tasks are mapped onto the nodes by passing the tasks as messages for meta-execution. A user program can be written to interface to a single virtual topology, which can then be automatically mapped onto whatever physical topology is offered by the hardware organization. There are several other constructs, such as *templates* and *ports*, which facilitate program creation, but do not present major implementation difficulties.

An issue related to task scheduling is task priority. Early concurrent logic languages specified that goals were required to be executed in a *fair manner*. Fairness is difficult to define in a manner that can be easily implemented. One weak definition is that all tasks which *can* execute *are* attempted at some time. This guarantees avoidance of spurious deadlock, i.e., deadlock *not* due to cyclic dependencies introduced by the programmer. Normally, tail-recursion optimization (TRO) is implemented wherein a selected body goal is directly executed and all others are wrapped up as goal records and enqueued. By extending the life of a thread through a selected child in this manner, efficient use of registers for argument passing can be achieved. Fair execution is emulated in a number of systems by a time-slicing technique, wherein every k reductions, TRO is replaced by enqueueing *all* body goals at the back of the queue, and switching in a goal from the front of the queue. Implementation incurs the overhead of updating a counter comparing it to k for each reduction, as well as enabling queue access from both the front and back.

The KL1 PIM systems took a different approach, discarding the notion of task execution fairness altogether. It is replaced by a goal priority scheme, wherein the scheduler makes its best effort to abide by priorities. This allows programming techniques such as speculative exploration of alternative solutions.[18] KL1 allows goal pragma that set priorities relative to a parent goal or a collection of goals called a *shoen*. These *logical* priorities, potentially ranging from 0 to 2^{32}, are retained in goal records, but also mapped into a smaller *physical* range for purpose of sorting. For example, if the physical range is 0 to 2^{14}, then the KL1 implementations use an array of 2^{14} queues. The non-empty queues are linked to allow efficient dequeueing across priorities. Insertion of a goal into an empty queue requires a linear search up to the nearest non-empty neighbors to update the links. This algorithm is sufficiently simple for its microcoded implementation, although software-based implementations might be better served by balanced, priority trees.

All the statistics given in Section 4.3 taken together indicate, among other things, that a not insignificant number of programs use "active" tasks, i.e., process groups are spawned to implement active objects that compute and communicate until the termination of the algorithm. For example, instead of implementing a heap data structure as a complex term to be passed as a procedure argument, the heap can be implemented as a group of node tasks connected by streams. Heap management algorithms proceed by message passing on these streams. This object-oriented programming style causes frequent suspensions because the processes composing the active objects are normally suspended, awaking only upon receiving a message upon a stream. Yet all the parallel systems previously mentioned implement *process-oriented scheduling* wherein a goal reduction leads to the enqueuing of its body goals

[18]An alternative method for enabling speculative search in CC languages proposes a guard that succeeds only if the executing processor is idle [37].

onto runtime work queues, with one of the goals selected for local execution (analogous to tail recursion optimization). Such a scheduling model executes active programs inefficiently.

Ueda and Morita proposed an alternative model called *message-oriented scheduling* [95, 96] for more efficient active program execution. The main idea is to transfer control to a stream consumer at the point when the producer sends the message. In the case where buffering can be avoided, this method of task switching to an active process has less overhead than the standard execution mechanism. Ueda and Morita implemented this method for shared-memory multiprocessors by scheduling from a global work pool. Since control is transferred immediately upon message sending, effectively independent chains of message sends are executed by the processors. Their initial performance results are extraordinarily good: naive reverse executes on a single Symmetry 80386 processor at 3.3 seconds (cf., jc (Janus) [39] runs at 3.1 seconds and Monaco (FGHC) [88] runs at 15.4 seconds[19]). Furthermore, almost linear speedups are achieved, as well as comparable performance to optimized 'C'. Ueda and Morita go further, comparing message-oriented to process-oriented systems on the VAX11/780. Three small benchmark programs achieved −300%, 40%, and 360% speedups using message-oriented execution, indicating that the idea is viable.

4.8 Granularity Control

Concurrent logic programs are fine grained: Alkalaj [3] measured from "20 to several hundred single-cycle instructions" per average goal reduction. Taylor [85] measured FCP granularity on a hypercube as a ratio of reductions/messages-passed,[20] ranging from 3.5–220. It is clear that granularity is very much dependent on application and programming style, but even in the best case, granularity is still low compared to conventional approaches to parallel programming in imperative languages.

The advantage of fine-grained concurrent languages is the abundance of potential parallelism. However, the main disadvantage is that too-fine granularity can lead to excess overheads in task management. Alkalaj [3] has shown that 50% of the execution time of large FCP applications is spent on goal management for a reasonable machine execution model. His recommendation was a specialized hardware organization to support this efficiently. Such directions are promising, as echoed for instance in the hardware implementations of threaded architectures, mainly predicated on dataflow languages (e.g., [69]). Special hardware or not, it is necessary to boost efficiency by "collecting" granularity at compile time [73, 92].

Ideas along these lines were developed for logic programs by Debray *et al.* [25], King and Soper [49], and Tick and Zhong [91]. Debray's design seeks to construct, at compile time, estimators of input argument size, and formulate these estimates into granularity estimations. At runtime, a granularity estimate is evaluated for each procedure invocation, and the estimated value is used to make dynamic scheduling decisions. For example, if the

[19]The latter two times were calibrated downwards from raw measurements (3.9 sec and 19.2 sec) made by the author on a 16 MHz Symmetry, since Ueda and Morita's measurements were made on a 20 MHz Symmetry.

[20]In a distributed memory multiprocessor implementation, messages would be used for communicating values down a stream from producer to consumer, for example.

weight is below a threshold, a task will not be spawned because of excessive overhead.

King [49] discusses an analysis technique with no runtime component. Similar to Debray's method, granularity is modeled as a function of argument size; however, these sizes are estimated by abstract interpretation. The analysis associates argument types in the concrete domain with a finite abstract domain of argument sizes. Another analysis suggested by King associates control structure in the concrete domain with a finite abstract domain of procedure complexities. The results are purely static determination of granularities. King uses these analyses, as an alternative to profiling for example, to drive task sequentialization (see Section 4.9).

Zhong's approach [91] attempts to remove the complexity of argument-size estimation (both at compile time and subsequent runtime evaluation costs) by introducing an abstract "iteration parameter" which is a proxy for relevant granularity information. The remainder of the scheme is similar to Debray's, with the major distinction that the estimators are easier to formulate, cheaper to evaluate, but far less accurate. Furthermore, the weights computed are relative, e.g., it can be estimated that one task is half the weight of another task, but it cannot be determined if either are below some absolute threshold weight.

The verdict is not yet in on the utility of these granularity analyses, because empirical data is sparse. Robust analyzers and larger benchmarks are needed.

4.9 Abstract Instruction Set Architectures

The Warren Abstract Machine (WAM) [2, 98] had a great influence on the various concurrent logic language implementations discussed in previous sections. The important differences among the abstract machines developed for committed-choice languages are in their storage models. The primary distinction is whether the heap is based in shared memory [20, 39, 43, 48, 51, 88], or distributed memory [54, 64, 75, 85]. In general, all variables and terms are stored in a heap, and memory is reclaimed by explicit, periodic garbage collection. Goals are usually represented by heap terms that can be linked into work queues for scheduling. A goal that is suspended can "float" on the heap, to be relinked to a work queue upon binding its hooked variable.

There are several variations to this basic model to gain efficiency. Goal records can be constrained to be fixed size and queued in free lists facilitating memory reuse. Furthermore, all data structures can be partitioned onto heaps corresponding to size, each with its own free list for ease of (de)allocation. Crammond [20] split arguments away from goal records, allocating them on their own stacks to improve locality and reuse. With arguments allocated separately from goal records, goal-record locality improves, and arguments no longer need to be of fixed size if allocated in a stack-based fashion. However, this results in the creation of "holes," i.e., deallocated frames trapped below the top of stack, which can require general garbage collection if they grow too large. Crammond [18] illustrates the extent of this problem for some small benchmarks.

In addition, bookkeeping structures for evaluating deep guards and suspension management are necessary. Recall from Section 4.4 that deep guards and sequential conjunctions require the use of environments which hold values of variables active throughout the clause try or sequential body evaluation. The environment is needed for sequential body evaluation because unlike goal-stacking implementations, depth-first sequential procedure evaluations requires environment stacking.

Practical implementations require at most one environment per invocation [20], which is deallocated upon body completion[21] or guard failure (to be reallocated for the next clause try). In addition, a *trail* is needed for atomic tell unification wherein failure and suspension during unification must "back out" all bindings generated. Suspension management requires a suspension stack holding pointers to input arguments that are needed but are as yet unbound.

A less important distinction among the systems are their abstract instruction sets. The instruction sets of the various machines follow the general WAM model, passing arguments through dedicated registers, and having a set of additional state registers for control and storage management. The instruction sets can be broken down into similar groups. Older models use WAM-like indexing control instructions, whereas decision-graph compilers for flat languages avoid shallow backtracking and much of the required control instructions. Head matching (ask unification) is compiled with **wait** instructions that will push their corresponding argument onto the suspension stack if it is not instantiated. Tell unification is compiled into **get** instructions that will make assignments or invoke a general unifier. Finally, body goals are generated with **put** instructions for loading arguments and enqueueing goal records. As mentioned in Section 4.7, usually a form of tail recursion optimization (TRO) can be implemented by loading the arguments of one of the body goals directly into the argument registers and jumping to the goal code.

Additional instructions are needed for the (de)allocation of local environments (for non-flat languages and/or sequential conjunctions) and heap storage. Goal-management instructions are responsible for terminating a thread (in unit clauses), enqueueing a goal (creating threads), directly executing a goal (TRO, called *promoting* a thread in JAM), and initiating deep guards. Note that flat languages have threads that live very short lives, not counting promotions. Sequential conjunctions, introduced in Parlog, do not lengthen threads in practice because they are necessarily implemented with trees of local environments (see Section 4.4). This stems from the fact that within a sequential conjunction, concurrent goals may execute. If total sequentialization of a goal and all its children can be specified or derived, then these local environments can be stacked, resulting in superior space and execution time efficiency. This would be a true elongation of threads, resulting in increased performance [50, 57]. Such an implementation requires a sequential call as well as stack (de)allocation instructions.

Arithmetic instructions and builtin predicates must be able to suspend if executed before commit, or be enqueued as bonefide goals if executed after commit. In a shared-memory multiprocessor where latencies are short, JAM optimizes this by checking arithmetic operator inputs in the body, and if available, executing the arithmetic in place. Otherwise, a goal is created. In distributed memory multiprocessors where nonlocal access latencies are long, it pays to spawn arithmetic goals in any case, as is done in threaded architectures for dataflow languages. Similarly, array accesses are spawned as independent goals. This is done even on shared-memory multiprocessors because static analysis of array indicies to

[21]More precisely, after all instructions, in the immediate thread, that access the environment, have been executed.

determine dependencies is very difficult [9].

A trend towards reduced abstract machine design, following the principles of RISC design, has led to instruction sets such as Carmel [41], SAM [29], the jc machine [39], Kliger's machine [51], and the various PIM architectures [54, 64, 75]. For example, Strand, FGHC, Fleng, Janus, and FCP(|) [77] sufficiently simplify the execution model, obviating trailing, environments, atomic tell unification, and a process hierarchy for deep guards and sequential conjuncts. This allows these compilers to concentrate on optimizations, such as decision-graph generation, in-line arithmetic, and global register allocation. Sequential implementations offer further performance gains, obviating locking and allowing the leverage of compilation into 'C'. Debray and Tick measured a mean speedup of 2.4 comparing jc with Monaco for six small benchmarks [88], illustrating the potential advantages of sophisticated register allocation, and streamlined binding mechanisms.

Readers interested in concurrent logic language instruction-set design are referred to Crammond [20], Foster and Taylor [28], and Kliger [51] for the most complete expositions.

4.10 Stream Communication, Arrays and Garbage Collection

A major defect in concurrent logic languages and their implementations is inefficient use of memory. This problem is prevalent in the treatment of communication streams and data arrays, and is exacerbated in distributed-memory multiprocessors. A general, after-the-fact, solution to the problem is the construction of ever more efficient garbage collectors, about which I comment at the end of this section. I first discuss pre-emptive solutions, such as making stream communication efficient with buffers and migration.

Streams are second-class citizens in most logic programming languages. Stream communication is programmed by having a producer write messages into a difference list, the head of which is read by a consumer. To nondeterminately merge multiple streams, a chain of active *merge* processes is needed. This methodology was stressed in the original literature because it is elegant and all that is offered at the *language* level. However, straightforward implementation of streams defined in this manner can be highly inefficient. First, merged streams incur extra process reductions, lengthening transmission *delay*. Second, naive stream merging can result in *unfair* data transmission. Third, if the memory cells comprising a stream and the reader of that stream are located on different processors in a distributed-memory system, then reading a value requires the overhead of sending a request message.[22]

The fairness problem can be solved with more sophisticated, dynamically-balanced merge trees [78], although this is expensive in time. The delay problem has been solved both in software and hardware. In software, a data type, called a *mutual reference*, interfaces multiple writers to a single reader [79]. Writing to one of the streams will atomically write the merged output stream and update the mutual reference to point to the new output tail. This scheme, originally designed for FCP, is the essence of implementing streams as buffers in other languages also. For example, the PIMs implement mergers, in microcode, in a similar manner [46, 94]. The critical difference is that the new data structure is hid-

[22]Analogous to driving all over town to pick up mail at different post offices, instead of having all mail delivered directly to your house.

den from the language definition. Furthermore, with MRB (discussed below), memory can potentially be reused when merging a new writer into the stream.

The indirection problem could be corrected by locating the buffer with the consumer (similar in intent to message-oriented scheduling, see Section 4.7); however, in most concurrent logic languages, multiple consumers are permitted, and single consumers are not recognized as such by the compiler. Global analysis might be used to determine single consumer streams, or the languages can be restricted. The latter solution is another deevolutionary step (notably Janus [71] and $\mathcal{A}'\mathcal{UM}$ [100]) with a "single writer/single reader" restriction and abstract stream semantics that constrain implementations to a lesser degree.

Janus defines a *bag* data type that can be used as a multiple-writer stream, with no constraint on write order (i.e., writers nondeterministically add items to the bag and order is not guaranteed). Janus also defines standard arrays; however, the restriction permits an implementation to automatically reuse array locations. Neither bags, nor reusable arrays, have yet been implemented for Janus. $\mathcal{A}'\mathcal{UM}$-90 is called a Stream-based Concurrent Object-Oriented programming Language (SCOOL) by its authors [53], emphasizing the first-class citizenship of streams. Streams are implemented as buffer objects that can migrate. The migration policy moves buffers to their (unique) consumers, thereby obviating the overhead of sending read requests. This is implemented by having the producer and consumer initially communicate with **where** and **here** messages, allowing them to locate each other and begin message copying. Future messages are forwarded automatically to the new location. For a generate-&-test prime-number generator, migration achieved a speedup of 12% (on 10 Symmetry processors) compared to no migration [53]. This increased to 70% speedup on two Sparcstations connected over an Ethernet.

The general topic of garbage collection is too large to cover here, but it is important nonetheless. There are fundamentally two approaches to garbage collection: static and dynamic. Static collection requires compiler analysis to determine the guaranteed reusability of a data structure. Code can then be generated directly for memory reuse. Incremental dynamic collection involves runtime checking to determine reusability of structures. Furthermore, dynamic garbage collections across an entire memory (local or global) are required when the previous incremental methods fail. Examples of these collection types within concurrent logic programming are abstract interpretation for local reuse [82], binary reference counting with multiple reference bits (MRB) [14, 67], and several stop & copy schemes (e.g., [35, 45, 46, 62]).

The MRB scheme [14] is a one-bit approximative reference count per data cell. If the flag is off, the cell can be reused because it is guaranteed to have a single reader. However, once the flag is set, it becomes stuck and no reuse is possible. Setting the flag requires nontrivial rules for many of the KL1 abstract instructions that manipulate memory. The advantages, however, can be significant, for example, array copying can be dynamically converted to destructive update by exploiting the MRB method. Nishida *et al.* [67] demonstrated the effectiveness of the MRB scheme for a shared-memory multiprocessor model. Depending on data cache configuration, two small benchmarks displayed bus traffic reduction from 20%–57% on 16 processors. The least beneficial result of 20% reduction clearly showed a drastic increase in cache-to-cache traffic that was indicative of reused cells being transferred between processors. Overall the method achieved significant reduction in memory-to-cache ("swap in") traffic, indicating success at improving locality by reuse.

Other types of static garbage collection for concurrent logic programs include "local reuse" techniques wherein reference information is collected at compile time and used to destructively update data objects at run time. Sundararajan *et al.* [82] describe one such analysis scheme and Foster and Winsborough [31] give an associated code generation method for local reuse of reclaimed cells. Essentially abstract reuse registers are used to cache pointers to dead objects which can subsequently be effectively reallocated. Gudjonsson and Winsborough [40] describe "update in place" analysis for Prolog, which can achieve even higher efficiency than the previous local reuse techniques. Essentially the performance gain is achieved by avoiding rewriting subterms in the dead object that are needed in the newly allocated object. However, it is not entirely clear if this scheme can be applied to concurrent logic programs.

Dynamic schemes still have utility because such static analyses usually have inaccuracies in order to guarantee that the information is conservative. This occurs primarily because of array indexing and inaccurate aliasing information. Further research is needed to empirically ascertain the practicality and accuracy of the static analyses.

The challenges of implementing efficient garbage collection schemes for concurrent logic, object-oriented, and functional programs are similar. There are several garbage collection schemes proposed (and some prototyped) for concurrent logic languages. These efforts have concentrated on stop & copy schemes for shared-memory multiprocessors, mirroring the general sophistication of the corresponding runtime systems. Research has recently focused on 1) distributed memory garbage collection schemes that do not require barrier synchronization of all processors within the collector [52]; 2) efficient, parallel stop & copy garbage collectors for shared memory [18, 45], and 3) generation-scavenging garbage collectors for reducing collection latency by interning long-lived objects [68].

For generation-scavenging schemes, an object that is assumed to have a long life is *interned* or cached in an additional space that is not involved in the standard two-space copying. A problem arises when the "old" space (the interned space) points into the "new" space because, garbage collection roots are not kept for old space objects and thus an object might miss being copied and erroneously become garbage. This can occur for interned logical variables and reused objects (e.g., via MRB) in the old space, pointing into the new space. Methods of trailing these unsafe cells (in the old space) and implementing indirection tables are costly [62], but seemingly unavoidable.

5 Summary

Concurrent logic programming languages have been deevolving since their inception, about ten years ago, because of the *tatonnement* that balances what systems designers and compiler writers can *supply* with what features applications writers *demand*. The implementation history traces a steady improvement in execution performance at the price of ever weakening language. This historical cycle between evolving and deevolving languages is not unique to logic programming: it was seen in Lisp moving into Scheme, as well as Algol moving into Pascal. The deevolution is positive in the sense that the shakedown is market driven, because you can't sell what you can't practically construct. Furthermore, recent research in combining "don't know" and "don't care" nondeterminism in Andorra-like systems represents an upward swing back to evolution, perhaps towards a Renessiance.

Acknowledgements

E. Tick was supported by an NSF Presidential Young Investigator award, with matching funds from Sequent Computer Systems Inc., and a grant from the Institute of New Generation Computer Technology (ICOT). I thank referees C. Palamidessi and K. Ueda, and the anonymous referees, for their particularly inciteful and plentiful comments. I also thank T. Chikayama, J. Crammond, and S. Debray for sharing their expert knowledge, greatly assisting in the completion of the article.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley, Reading MA, 1985.

[2] H. Ait-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* MIT Press, Cambridge, MA, 1991.

[3] L. Alkalaj, T. Lang, and M. Ercegovac. Architectural Support for the Management of Tightly-Coupled, Fine-Grain Goals in Flat Concurrent Prolog. In *International Symposium on Computer Architecture*, pages 292–301, Seattle, June 1990. IEEE Computer Society Press.

[4] Arvind and R. A. Iannucci. Two Fundamental Issues in Multiprocessing. In *DFVLR: Conference on Parallel Processing in Science and Engineering.* Bonn-Bad Godesberg, June 1987.

[5] Arvind, R. S. Nikhil, and K. K. Pingali. I-Structures: Data Structures for Parallel Computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632, October 1989.

[6] S. Asano, S. Isobe, and H. Sakai. The Unique Features of PIM/k: A Parallel Inference Machine with Hierarchical Cache System. Technical Report TR-767, ICOT, 1-4-28 Mita, Minato-Ku Tokyo 108, Japan, April 1992.

[7] R. Bahgat. *Pandora: Non-deterministic Parallel Logic Programming.* World Scientific Publishing Co., Singapore, 1993.

[8] 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.

[9] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Norwell MA, 1988.

[10] L. Bougé. On the Existence of Symmetric Algorithms to Find Leaders in Networks of Communicating Sequential Processes. *Acta Informatica*, 25:179–201, 1988.

[11] C. Chandy and S. Taylor. *An Introduction to Parallel Programming.* Jones and Bartlett, Boston MA., 1991.

[12] A. Cheese. *Parallel Execution of Parlog.* Number 586 in Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1992.

[13] T. Chikayama. A Portable and Efficient Implementation of KL1. In *International Symposium on Programming Language Implementation and Logic Programming*, Madrid, September 1994. Submitted.

[14] T. Chikayama and Y. Kimura. Multiple Reference Management in Flat GHC. In *International Conference on Logic Programming*, pages 276–293. University of Melbourne, MIT Press, May 1987.

[15] K. L. Clark and S. Gregory. A Relational Language for Parallel Programming. In *Conference on Functional Programming Languages and Computer Architecture*, pages 171–178, Portsmouth NH, October 1981. ACM Press.

[16] V. S. Costa, D. H. D. Warren, and R. Yang. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-Parallelism. In *SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 83–93. Williamsburg, ACM Press, April 1991.

[17] 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.

[18] J. A. Crammond. *Implementation of Committed-Choice Logic Languages on Shared-Memory Multiprocessors.* PhD thesis, Heriot-Watt University, Endinburgh, May 1988.

[19] J. A. Crammond. Scheduling and Variable Assignment in the Parallel Parlog Implementation. In *North American Conference on Logic Programming*, pages 642–657. Austin, MIT Press, October 1990.

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

[21] D. E. Culler, A. Sah, K. E. Schauser, T. von Eicken, and J. Wawrzynek. Fine-Grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 164–175, Santa Clara, April 1991. IEEE Computer Society Press.

[22] F. de Boer and C. Palamidessi. Concurrent Logic Languages: Asynchronism and Language Comparison. In *North American Conference on Logic Programming*, pages 175–194. Austin, MIT Press, October 1990.

[23] F. de Boer and C. Palamidessi. Embedding as a Tool for Language Comparison. *Information and Computation*, 1993.

[24] S. Debray, S. Kannan, and M. Paithane. Weighted Decision Trees. In *Joint International Conference and Symposium on Logic Programming*, pages 654–668. Washington D.C., MIT Press, November 1992.

[25] S. K. Debray, N.-W. Lin, and M. Hermenegildo. Task Granularity Analysis in Logic Programs. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 174–188, White Plains, NY, June 1990. ACM Press.

[26] I. Foster. *Systems Programming in Parallel Logic Languages.* Prentice Hall, Englewood Cliffs, NJ, 1990.

[27] I. Foster. Information Hiding in Parallel Programs. Technical Report MCS-P290-0292, Argonne National Laboratory, 1992.

[28] I. Foster and S. Taylor. Flat PARLOG: A Basis for Comparison. *International Journal of Parallel Programming*, 16(2):87–125, 1987.

[29] I. Foster and S. Taylor. Strand: A Practical Parallel Programming Language. In *North American Conference on Logic Programming*, pages 497–512. Cleveland, MIT Press, October 1989.

[30] I. Foster and S. Taylor. A Compiler Approach to Scalable Concurrent Program Design. Technical Report MCS-P306-0492, Argonne National Laboratory, 1992.

[31] I. Foster and W. Winsborough. Copy Avoidance through Compile-Time Analysis and Local Reuse. In *International Symposium on Logic Programming*, pages 455–469. San Diego, MIT Press, November 1991.

[32] M. Furuichi, K. Taki, and N. Ichiyoshi. A Multi-Level Load Balancing Scheme for OR-Parallel Exhaustive Search Programs on the Multi-PSI. In *SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 50–59, Seattle, March 1990. ACM Press.

[33] H. Gaifman, M. J. Maher, and E. Shapiro. Reactive Behaviour Semantics for Concurrent Constraint Logic Programs. In *North American Conference on Logic Programming.* Cleveland, MIT Press, October 1989.

[34] R. Gerth, M. Codish, Y. Lichtenstein, and E. Shapiro. Fully Abstract Denotational Semantics for Concurrent Prolog. In *Third Annual IEEE Symposium on Logic in Computer Science*, pages 320–335. IEEE Computer Society Press, 1988.

[35] A. Goto, Y. Kimura, T. Nakagawa, and T. Chikayama. Lazy Reference Counting: An Incremental Garbage Collection Method for Parallel Inference Machines. In *International Conference and Symposium on Logic Programming*, pages 1241–1256. University of Washington, MIT Press, August 1988.

[36] S. Gregory. *Parallel Logic Programming in PARLOG: The Language and its Implementation.* Addison-Wesley Ltd., Wokingham, England, 1987.

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

[38] S. Gregory, I. Foster, A. Burt, and G. Ringwood. An Abstract Machine for the Implementation of Parlog on Uniprocessors. *New Generation Computing*, 6:389–420, 1989.

[39] D. Gudeman, K. De Bosschere, and S. K. Debray. jc: An Efficient and Portable Sequential Implementation of Janus. In *Joint International Conference and Symposium on Logic Programming*, pages 399–413. Washington D.C., MIT Press, November 1992.

[40] G. Gudjonsson and W. Winsborough. Update In Place: Overview of the Siva Project. In *International Logic Programming Symposium*, pages 94–113, Vancouver B.C., October 1993. MIT Press.

[41] A. Harsat and R. Ginosar. CARMEL-4 The Unify-Spawn Machine for FCP. In *International Conference on Logic Programming*, pages 840–854. Paris, MIT Press, June 1991.

[42] M. Hirata. Programming Language Doc and its Self-Description, or, X=X is Considered Harmful. In *Proceedings of the 3^{rd} Conference of Japan Society of Software Science and Technolog*, pages 69–72, Tokyo, 1986.

[43] A. Houri and E. Y. Shapiro. A Sequential Abstract Machine for Flat Concurrent Prolog. In E. Y. Shapiro, editor, *Concurrent Prolog: Collected Papers*, volume 2, pages 513–574. MIT Press, Cambridge MA, 1987.

[44] N. Ichiyoshi and K. Kimura. Asymptotic Load Balance of Distributed Hash Tables. In *International Conference on Fifth Generation Computer Systems*, pages 869–876, Tokyo, June 1992. ICOT.

[45] A. Imai and E. Tick. Evaluation of Parallel Copying Garbage Collection on a Shared-Memory Multiprocessor. *IEEE Transactions on Parallel and Distributed Computing*, 4(9):1030–1040, September 1993.

[46] Y. Inamura, N. Ichiyoshi, K. Rokusawa, and K. Nakajima. Optimization Techniques Using the MRB and Their Evaluation on the Multi-PSI/V2. In *North American Conference on Logic Programming*, pages 907–921. Cleveland, MIT Press, October 1989.

[47] S. Janson and S. Haridi. Programming Paradigms of the Andorra Kernel Language. In *International Symposium on Logic Programming*, pages 167–183. San Diego, MIT Press, November 1991.

[48] 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.

[49] 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.

[50] 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.

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

[52] H. Koike and H. Tanaka. Generation Scavenging GC on Distributed-Memory Parallel Computers. In *Proceedings of High Performance and Parallel Computing in Lisp.* EUROPAL Workshop, London, November 1990.

[53] K. Konishi, T. Maruyama, A. Konagaya, K. Yoshida, and T. Chikayama. Implementing Streams on Parallel Machines with Distributed Memory. In *International Conference on Fifth Generation Computer Systems*, pages 791–798, Tokyo, June 1992. ICOT.

[54] K. Kumon, A. Asato, S. Arai, T. Shinogi, A. Hattori, H. Hatazawa, and K. Hirano. Architecture and Implementation of PIM/p. In *International Conference on Fifth Generation Computer Systems*, pages 414–424, Tokyo, June 1992. ICOT.

[55] G. Lindstrom. Functional Programming and the Logic Variable. In *SIGPLAN Symposium on Principles of Programming Languages*, pages 266–280. New Orleans, ACM Press, 1985.

[56] M. J. Maher. Logic Semantics for a Class of Committed-Choice Programs. In *International Conference on Logic Programming*, pages 858–876. University of Melbourne, MIT Press, May 1987.

[57] 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.

[58] M. Meier. Better Late Than Never. In E. Tick and G. Succi, editors, *Implementations of Logic Programming Systems.* Kluwer Academic Publishers, 1994.

[59] C. Mierowsky *et al.* The Design and Implementation of Flat Concurrent Prolog. Technical Report CS85-09, The Weizmann Institute of Science, Rehovot, Israel, July 1985.

[60] T. Miyazaki, A. Takeuchi, and T. Chikayama. A Sequential Implementation of Concurrent Prolog Based on the Shallow Binding Scheme. In E. Y. Shapiro, editor, *Concurrent Prolog: Collected Papers*, volume 2, pages 496–512. MIT Press, Cambridge MA., 1987.

[61] T. Nakagawa, N. Ido, T. Tarui, M. Asaie, and M. Sugie. Hardware Implementation of Dynamic Load Balancing in the Parallel Inference Machine PIM/c. In *International Conference on Fifth Generation Computer Systems*, pages 723–730, Tokyo, June 1992. ICOT.

[62] K. Nakajima. Piling GC: Efficient Garbage Collection for AI Languages. In *IFIP Working Conference on Parallel Processing*, pages 201–204. Pisa, North Holland, May 1988.

[63] H. Nakashima and K. Nakajima. Hardware Architecture of the Sequential Inference Machine: PSI-II. In *International Symposium on Logic Programming*, pages 104–113. San Francisco, IEEE Computer Society Press, August 1987.

[64] H. Nakashima, K. Nakajima, S. Kondo, Y. Takeda, Y. Inamura, S. Onishi, and K. Masuda. Architecture and Implementation of PIM/m. In *International Conference on Fifth Generation Computer Systems*, pages 425–435, Tokyo, June 1992. ICOT.

[65] R. S. Nikhil, G. M. Papadopoulos, and Arvind. *T: A Multithreaded Massively Parallel Architecture. In *International Symposium on Computer Architecture*, pages 156–167. Gold Coast, IEEE Computer Society Press, May 1992.

[66] M. Nilsson and H. Tanaka. FLENG Prolog—The Language which turns Supercomputers into Parallel Prolog Machines. In E. Wada, editor, *Proceedings of the Logic Programming Conference*, number 264 in Lecture Notes in Computer Science, pages 170–179. Springer-Verlag, June 1986.

[67] K. Nishida, Y. Kimura, A. Matsumoto, and A. Goto. Evaluation of MRB Garbage Collection on Parallel Logic Programming Architectures. In *International Conference on Logic Programming*, pages 83–95. Jerusalem, MIT Press, June 1990.

[68] T. Ozawa, A. Hosoi, and A. Hattori. Generation Type Garbage Collection for Parallel Logic Languages. In *North American Conference on Logic Programming*, pages 291–305. Austin, MIT Press, October 1990.

[69] G. M. Papadopoulos and D. E. Culler. Monsoon: an Explicit Token-Store Architecture. In *International Symposium on Computer Architecture*, pages 82–91. Seattle, IEEE Computer Society Press, May 1990.

[70] H. Pinto. Implementing Meta-Interpreters and Compilers for Parallel Logic Languages in Prolog. Master's thesis, University of Edinburgh, Artificial Intelligence Applications Institute, September 1986.

[71] 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.

[72] V. A. Saraswat, M. Rinard, and P. Panangaden. Semantics Foundations of Concurrent Constraint Programming. In *SIGPLAN Symposium on Principles of Programming Languages*. ACM Press, 1991.

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

[74] M. Sato and A. Goto. Evaluation of the KL1 Parallel System on a Shared Memory Multiprocessor. In *IFIP Working Conference on Parallel Processing*, pages 305–318. Pisa, North Holland, May 1988.

[75] M. Sato, K. Kato, K. Takeda, and T. Oohara. Exploiting Fine Grain Parallelism in Logic Programming on a Parallel Inference Machine. Technical Report TR-676, ICOT, 1-4-28 Mita, Minato-ku Tokyo 108, Japan, August 1991.

[76] E. Y. Shapiro. A Subset of Concurrent Prolog and Its Interpreter. In E. Y. Shapiro, editor, *Concurrent Prolog: Collected Papers*, volume 1, pages 27–83. MIT Press, Cambridge MA., 1987.

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

[78] E. Y. Shapiro and C. Mierowsky. Fair, Biased, and Self-Balancing Merge Operators: Their Specification and Implementation in Concurrent Prolog. In E. Y. Shapiro, editor, *Concurrent Prolog: Collected Papers*, volume 2, pages 392–413. MIT Press, Cambridge MA., 1987.

[79] E. Y. Shapiro and S. Safra. Multiway Merge with Constant Delay in Concurrent Prolog. In E. Y. Shapiro, editor, *Concurrent Prolog: Collected Papers*, volume 2, pages 416–420. MIT Press, Cambridge MA., 1987.

[80] W. Silverman, M. Hirsch, A. Houri, and E. Y. Shapiro. The Logix System User Manual, Version 1.21. In E. Y. Shapiro, editor, *Concurrent Prolog: Collected Papers*, volume 2, pages 46–77. MIT Press, Cambridge MA, 1987.

[81] M. Sugie, M. Yoneyama, N. Ido, and T. Tarui. Load-Dispatching Strategies on Parallel Inference Machines. In *International Conference on Fifth Generation Computer Systems*, pages 987–993, Tokyo, November 1988. ICOT.

[82] R. Sundararajan, A. V. S. Sastry, and E. Tick. Variable Threadedness Analysis for Concurrent Logic Programs. In *Joint International Conference and Symposium on Logic Programming*, pages 493–508. Washington D.C., MIT Press, November 1992.

[83] A. Takeuchi. *Parallel Logic Programming.* John Wiley & Sons, Inc., New York, 1992.

[84] K. Taki, K. Nakajima, H. Nakashima, and M. Ikeda. Performance and Architectual Evaluation of the PSI Machine. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 128–135. Palo Alto, IEEE Computer Society Press, October 1987.

[85] S. Taylor. *Parallel Logic Programming Techniques.* Prentice Hall, Englewood Cliffs, NJ, 1989.

[86] E. Tick. *Memory Performance of Prolog Architectures.* Kluwer Academic Publishers, Norwell MA., 1987.

[87] E. Tick. Appraisal of Parallel Processing Research at ICOT. *Future Generation Computer Systems*, 9(2):127–136, 1993.

[88] 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.

[89] E. Tick and J. A. Crammond. Comparison of Two Shared-Memory Emulators for Flat Committed-Choice Logic Programs. In *International Conference on Parallel Processing*, volume 2, pages 236–242, Penn State, August 1990.

[90] 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. In press.

[91] E. Tick and X. Zhong. A Compile-Time Granularity Analysis Algorithm and its Performance Evaluation. *New Generation Computing*, 11(3–4):271–295, June 1993.

[92] 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.

[93] R. Trehan. A Comparison of Committed Choice Non-Determinate Logic Languages Parallelism in a Mathematical Equation Solver (PRESS). Master's thesis, University of Edinburgh, Artificial Intelligence Applications Institute, September 1986.

[94] K. Ueda and T. Chikayama. Efficient Stream/Array Processing in Logic Programming Language. In *International Conference on Fifth Generation Computer Systems*, pages 317–326, Tokyo, 1984. ICOT.

[95] 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.

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

[97] P. Wadler. Strictness Analysis on Non-Flat Domains (by Abstract Interpretation Over Finite Domains). In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, pages 181–198. Ellis Horwood Ltd, Chichester, 1987.

[98] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, Artificial Intelligence Center, SRI International, 333 Ravenswood Ave, Menlo Park CA 94025, 1983.

[99] K. Yanoo. An Optimizing Compiler for a Parallel Inference Language. In H. Tanaka, editor, *Annual Report of the Research on Parallel Inference Engine*, pages 71–94. University of Tokyo, April 1992. (in Japanese).

[100] K. Yoshida and T. Chikayama. A'UM: A Stream-Based Object-Oriented Language. In *International Conference on Fifth Generation Computer Systems*, pages 638–649, Tokyo, November 1988. ICOT.