Workshop on Parallel Logic Programming and its Programming Environments

Takashi Chikayama — ICOT[†]
Evan Tick — University of Oregon
Editors

CIS-TR-94-04
March 1994

Abstract

The *NSF/ICOT Workshop on Parallel Logic Programming and its Programming Environments* was held at the University of Oregon on March 4–6 1994. The primary goal of the workshop was the exchange of research results in the important areas of concurrent symbolic language implementation, and environments to aid program development. About forty international researchers attended the workshop, giving over twenty invited technical presentations. The proceedings of these talks are compiled here: topics include compilation, runtime systems, environments, concurrent constraints, and execution models.

This workshop was funded through the generosity of the Institute for New Generation Computer Technology (ICOT) and the National Science Foundation (NSF grant 9403574).

[†]Institute for New Generation Computer Technology, 1–4–28 Mita, Minato-ku Tokyo 108, Japan

DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE
UNIVERSITY OF OREGON

Table of Contents

A Sequential Implementation of a Concurrent Constraint Language

David Gudeman

Abstract: We have been developing an optimizing compiler for a concurrent constraint programming language descended from Janus. The compiler has been developed under two guiding principles: first, that a given program should not have to pay the performance penalty for advanced features that are not used in the program, and second, that we will concentrate on generating good code locally before resorting to global analyses. The results have been good so far: our performance is close to that of C for programs that concentrate on lists and integer arithmetic. The discussion will concentrate on the various optimization strategies that have been used to achieve the performance we have, and on analyzing how much gain we get from each strategy.

Begin page numbering here.

(At number one, of course ☺)

(Target page numbers are written on the back of title pages in pencil.)

Implementing Selective Communication for Shared Memory Multiprocessors

Greg Morrisett

Abstract: Languages such as CSP, Ada, Occam, and more recently, Concurrent ML (CML) and Facile provide selective communication whereby a process can non-deterministically choose to synchronize and/or communicate with one of a set of processes. Most of these languages, with the notable exceptions of CML and Facile, place severe restrictions on the forms of communication in which a process can participate, because general selective communication is difficult to implement correctly and efficiently. For example, CSP allow receivers to choose from a set of alternative senders, but does not allow senders to choose from a set of receivers.

Recently, Knabe has derived a protocol for implementing the general selective communication primitives of Facile in a distributed environment. The formulation of the protocol makes communication channels active entities that coordinate selection. This approach makes understanding and reasoning about the protocol relatively simple, but the protocol itself is quite expensive. A minimum of three round-trip messages must be sent between the communicating processes and the channel and in practice, other messages must be sent to affect the communication.

On a shared memory multiprocessor, directly implementing Knabe's protocol can lead to disasterous performance when compared to a standard implementation of non-selective communication. We show how to specialize Knabe's protocol for the case of a shared memory multiprocessor in such a way that channels remain passive entities (i.e., data structures) and good performance is retained.

Partitioning Non-Strict Functional Languages

Satyan R. Coorg

Abstract: Partitioning is an important stage in compiling non-strict languages to conventional hardware. The goal of partitioning is to compile a non strict program to produce executable "threads". It is desirable to have large threads, as this amortizes the overhead of executing a thread over many instructions. Previous results produce "safe" partitions from non-strict programs, but fail to capture dependencies across recursive procedure boundaries. In this talk, I present the basic algorithm for partitioning and some preliminary results using abstract interpretation techniques which allow dependencies of recursive functions to be captured. I also describe a way of integrating the dependency information into the partitioning algorithm.

The Evaluation of Parallel Inference Machines

Kouichi Kumon

Abstract: This paper describes the performance measurement of KL1 processing system, especially for a single processing element of Parallel Inference Machine (PIM) using a set of small programs. All programs are run without any modifying the source codes. From the experimental results of the benchmarking, we found that historically used append RPS is not suitable to represent sytem performance of PIM, because in some machines, append RPS shows too optimal value compared to other benchmarks. The difference between append RPS and the other programs are presented. Also, the method to representative programs from a number of programs are shown.

ICOT has made five models of Parallel Inference Machine for KL1. Formerly, independent performance measurements of each machine has been make by researchers of their manufactures or ICOT, but no common comparison among PIMs has been made. To utilize the experience of making various types of PIM, the evaluation of all PIMs on the same environment is needed.

To obtain the best performance for a specific system, one can rewrite programs for that system, but in this experiment, we use same program to all systems without modification. The reason is that a system should absorb the difference of a coding manner of users, and should optimize these codes. Of course large programs like real application programs are the best for benchmarking, but we hope that the collections of small benchmark written by different programmers are the best for benchmarking. We hope that the collections of small benchmark programs supplement the smallness of the programs.

Unfortunately, there is no common benchmark program set for concurrent logic languages, like Specmark or Linpac in conventional machines. Historically, append RPS is used to represent the performance of a Prolog processing which is also used for KL1, but the execution of append does not use the important features, concurrency, that makes the KL1 distinct from the Prolog.

Currently, performance measurement for some system configurations from a single processor to a single cluster are performed. In this paper, we show some experimental results of a single processing element of PIM and KLIC system on Sparc.

Fully Demand-Driven Execution of Committed-Choice Programs

Bart Massey

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

A Portable and Efficient Implementation of KL1

Takashi Chikayama Tetsuro Fujise
ICOT

Daigo Sekita
Mitsubishi Research Institute

Abstract

An implementation scheme of a concurrent logic programming language KL1 by compiling it into C language code is investigated. The implementation can be fully portable with this scheme, as C compilers can be found on almost any computer systems in the market.

A feature called *generic objects* was introduced to the scheme, which allows adding new data types and their manipulation without even slightly changing the core implementation. It can provide foreign language interface much more flexible than conventional subroutine interfaces. Parallel implementations can also be built upon it.

An experimental implementation with the proposed scheme called KLIC was built to verify the design. Its sequential core showed reasonable efficiency in both time and space aspects; about twice as fast as the native code generated by SICStus or Aquarius Prolog with code size 30% smaller than SICStus native code or only 25% larger than SICStus abstract machine code.

1 Introduction

A concurrent logic programming language KL1[12] was chosen as the interface between the hardware and software research in the Japanese Fifth Generation Computer Systems project. The language has been proved to be a practical tool of parallel processing software research through the development of the operating system PIMOS[2] and various application software on parallel inference machines, Multi-PSI [11] and PIM [3].

Such implementations, however, have a serious disadvantage that they are not portable; although they are efficient, they run only on specially devised hardware. A portable byte-code implementation exists, but with limited performance.

To solve the problem, a scheme that allows very much portable implementation of compiling into C was investigated. Language C was chosen as the target language, as excellent optimizing compilers for the langauge are widely available nowadays. There are merits and demerits of using C as an intermediate language. We have designed an implementation scheme which detours the demerits and built an experimental system named KLIC.

The sequential core of KLIC showed reasonable efficiency in both time and space aspects. It ran about twice as fast as the native code generated by SICStus Prolog or Aquarius Prolog for representative benchmark programs on SparcStation 10 model 30. The code size became larger than abstract machine code but was found to remain reasonable; 30% smaller than native machine code and only 25% larger than abstract machine code generated by SICStus.

The structure of this paper is as follows. The merits of using C as an intermediate language, difficulties in constructing efficient C programs corresponding to KL1 programs, and our solutions to the problems are given in the next section. An outline of the experimental implementation is drawn in section 3. One of the key features called *generic* *objects* is described in secion 4. Finally, performance evaluation results of the sequential core are given in 5 followed by concluding remarks.

2 C as the Intermediate Language

We decided to use the language C as the intermediate language to implement KL1. This section first points out merits of the scheme. Then difficulties in obtaining reasonable efficiency with such a scheme are then investigated. Finally, our solutions to the efficiency problems are described.

2.1 Merits of Compilation into C

There are various merits in the scheme of compiling into C, in which most important ones are the following.

Portability The greatest merit is that the implementation can be quite portable. Porting KLIC to systems with a C compiler requires only adjusting some switches and recompiling.

Low-Level Optimization Some C compilers provide very good low-level optimization. By letting C compiler take care of low-level issues, the language implementation can concentrate on higher-level optimization issues.

Linkage with Programs in Other Languages Linking KL1 programs with programs written in C becomes quite easy. In addition, in Unix-like systems where C is "the" language most other language systems provide certain interface for. Thus, KL1 programs can also be linked with programs in almost any languages without much effort.

2.2 Efficiency Problems

Although compiling into C has the above-mentioned merits, it is not easy to realize reasonably efficient implementation for languages with an execution model quite different from C, such as KL1. Typical efficiency problems are as follows.

Costly function calls The language C and its implementations are designed having in mind that functions are not too small. Although function invocation and parameter passing overheads themselves may be small, dividing programs into functions makes program analysis more difficult, often resulting in less efficient object code. Predicates of KL1 are usually very small, often as small as one line of C code, many of them being recursive, prohibiting inline expansion. Thus, naive strategies such as compiling each KL1 predicate into one function may result in quite inefficient code.

Inability to control register allocation Certain global data, such as the free memory top pointer, are accessed very frequently. It would be best to keep such data on some dedicated registers during the whole execution. In most C implementations, however, such control of register allocation is not possible.[1]

[1]Some compilers (such as gcc) allow this, but using the feature is disadvantageous for portability; portability of reasonable efficiency is also an important part of the "portability" we need.

Cost of provision for interrupts Multi-processor implementations should process interrupts from other processors. Interrupts may require allocating memory, enqueueing goals or giving values to variables. Data accessed in these operations are frequently referenced and altered also in normal processing within a processor. Thus, certain locking on data or inhibition of interrupts may be required. These are quite costly with conventional operating systems.

Large object code size When logic programming language programs are compiled into native machine code, the object code size tends to become large. Increased working set sometimes results in performance worse than an abstract machine interpreter. When compiled through C, this code size increase might be amplified. Using runtime subroutines may reduce the code size but also reduces the speed.

2.3 Solutions

Our solutions to the efficiency problems described above are as follows.

One module as one function To avoid function call costs, compiling the whole program into one function may be the best. However, this will prevent separate compilation, which will be a serious problem with programs of practical size, especially during their development phase.

Our solution is to let the user control the size of compilation. One "module" that defines a set of closely related predicates is compiled into one C function. As far as predicates within the same module are calling one another, control transfer is by **goto** and arguments can be passed through local variables, which might be allocated on machine registers.

Caching global variable To make accesses to crucial global variables with lowest cost, such variables are cached on local variables. C compilers may allocate them on registers; even if not, accesses to local variables are less costly than to global variables with most modern processor architectures.

With caching, global variables may not always have the up-to-date value. We have to have this in mind in writing interrupt handlers. Even for synchronized runtime subroutines, passing all the cached variables to them and also having them passed back is quite costly.[2] Fortunately, we could design the maintenance principles of such variables so that only small numbers of them are to be passed and returned in most cases.

Efficient synchronization with interrupt handling Signal handlers are made to set a certain flag, which will be examined at certain timing convenient for normal processing. This synchronizes interrupt handling with normal processing without expensive interrupt inhibition.

In our system, this flag check is combined with another mandatory check: heap overflow check for garbage collection and thus made virtually costless. See the following sections for details.

Runtime routines for exceptional cases Fortunately, input/output modes of variable references can be decided much more easily during compilation for KL1 than for Prolog. By our deliberate design of task allotment between compiled code and runtime

[2]Passing back is especially inefficient, as only one result can be returned from a C function.

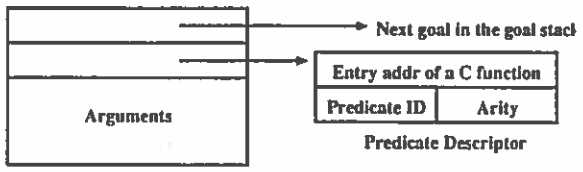


Figure 1: Goal Record

routines, common cases are handled in-line while exceptional cases are handled by the runtime. Object code size was thus kept reasonable without significantly slowing down the execution.

3 Experimental Implementation

This section describes the sequential core of our experimental implementation, KLIC. Parallel implementations now under development are built on this sequential core [8] [9].

3.1 Data Representation

Every KL1 term is represented with a `long int` word, with typically 32 bits. The lowest 2 bits are used for basic data type tags. The rest represents data: an address or immediate data. The 2 tag bits distinguish the following four basic types.

Variable Reference: The data part has the address of a variable value cell. Uninstantiated variables are represented as a self-reference pointer as in WAM[14]. Variables with goals awaiting its value will be described in the next section.

Atomic Data: For atomic data, 2 more lowest bits of the data part are used as tag extension, which distinguish symbolic atom, integer, etc. The remaining bits (typically 28 bits) represent the value.

Cons: The data part has the address of a two-word memory block for the cons cell.[3]

Functor: The data part has the address of a memory block of a functor structure. The lowest bit of the first word of the block tells functors from generic data objects.[4] For functors, the word contains a functor identifier. The rest is used for arguments.

3.2 Goal Management

Goal Stacks Ready goals are represented as goal records in the heap area and put into a LIFO goal stack, which is a linked list of goal records. In each reduction step, the topmost goal is popped up from the stack, reduced according to the program, and resultant children goals, if any, are pushed back to the stack.

The second field of a goal record has a pointer to a predicate descriptor that contains a pointer to the code (a C function) corresponding to the module it belongs to, the predicate ID within the module (an integer), and the number of arguments. The rest contains arguments of the goal (Figure 1).

[3]Cdr field comes first; cdr is more likely to be initiated as an uninstantiated variable and the heap top pointer itself can be used for its initiation if the first field is cdr, rather than car.

[4]Representation of generic objects will be described in section 4.

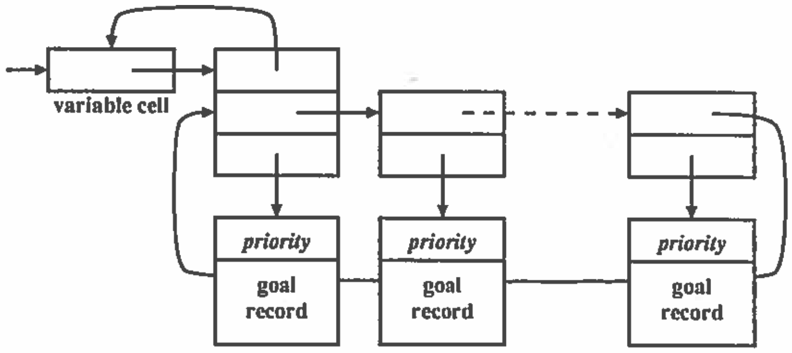


Figure 2: Suspended Goals

There can be multiple goal stacks corresponding to priority levels. The priority mechanism has been found to be very useful through application software research on PIM systems in describing various algorithms with speculative computation, and now is an indispensable feature of KL1.

The last entry in the goal stack is a sentinel goal which will schedule the goal stack with the next highest priority. The lowest priority goal stack has the sentinel goal for termination detection.

Goal records are allocated in the heap with other data. This allows allocation of variable cells within goal records, which reduces the working set. On the other hand, we abandoned incremental reusing of goal records and increase garbage collection overhead.

Suspended Goals A variable with goals awaiting for its instantiation is also represented by a pointer with the variable reference tag (figure 2). The pointer part references a *suspension record* that starts a list of hook records. The first word of the suspension record contains a pointer back to the variable, making a two-word loop.[5]

The rest of the suspension record contains a "hook record" that records a goal waiting for the variable's value. Hook records, including the first one within in the suspension record, are linked by their first words to form a loop. This loop structure allows efficient unification of two variables each with many suspended goals.

The first field of goal records for suspended goals are used to keep their priority values, instead of linkage for goal stack. The lowest bit of the field is set for suspended goals (see below for the reason).

A goal awaiting for the value of one of a set of variables is referenced from multiple hook records. When the goal is awaken through instantiation of one of the variables, the goal is put back to the goal stack. As the first word of the goal record is used for chaining the goals in the stack, the lowest bit of the word is naturally cleared. On instantiation of other varibles in the set, already awaken goals can be told by the bit.

3.3 Heap Area Management

The heap area is organized as shown in figure 3. Memory allocations are usually made at the heap pointer downwards. At the end of each reduction, whether the heap pointer is compared with the heap limit. If it points below the limit, the garbage collector will

[5]To know whether a variable is instantiated or not, dereference it twice and see whether that resulted in the same pointer. This works for both one- and two-word loops. The idea is originated by Hiroshi Nakashima.

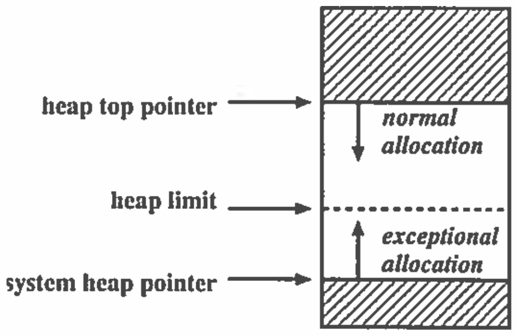


Figure 3: Organization of Heap Area

be called. The garbage collector adopts the copying scheme with certain modifications to cope with directly referenced structure elements.

Allocations in runtime subroutines, such as those implementing built-in predicates, are made at the system heap pointer upwards. By this, passing and returning the heap top pointer are avoided.

When allocations are made at the bottom, not only the system heap pointer but also the heap limit pointer is moved upwards by the same amount, always keeping a certain gap between the system heap pointer and the heap limit. The size of the gap is kept greater than the maximum memory allocation in one goal reduction.

3.4 Interrupt Handling

On multi-processor implementations, normal goal reductions are interrupted by other processors. When some interrupt (actually a *signal* in Unix) takes place, the signal handler will set a flag in a global variable and, at the same time, modifies the heap limit so that the check at the end of the current reduction will find it. The check routine can examine the flag and tell whether a garbage collection and/or interrupt handling are needed. Thus, no extra check of interrupts is required during normal processing, except that the heap limit has to be loaded from memory each time.

The same mechanism can be used to notify newly available goals with higher priority.[6] It is also used by the stepping tracer described below.

3.5 Compilation

The KL1 to C compiler of the current version is written in Prolog. It performs simple optimizations such as clause indexing but no global analysis whatsoever is made yet.

One module is compiled into one function of C, which has the following parts.

- Dispatch code to the predicate code for the topmost goal in the goal stack.
- Statement blocks corresponding to predicates defined in the module, each of which contains the following.
  - Statements to load arguments to local variables.[7]
  - Statements to index a clause.

[6]This can happen during unification when such a goal became ready by variable instantiation.

[7]If all the predicates in the module have more than, say, N arguments, loading first N arguments are done before predicate dispatch.

  - Statements corresponding to the bodies, including memory allocation, unification, goal creation.

    When a predicate within the module is to be called next, arguments are set up in local variables, heap overflow is checked, and the control is transferred to the top of the clause indexing code of the predicate. Otherwise, when the next goal to be executed is outside of the module or when no body goals exist, the control transfers to the last part of the function.
  - Statements for suspension and failure, jumped in from the indexing code.

- Statements for non-tail-recursive cases. First, heap overflow is checked and the garbage collector and/or the interrupt handler will be invoked if needed. Then, the topmost goal in the goal stack is examined, and, if it is for a predicate in the same module, the control is transferred back to the predicate dispatch code.

- Statements to return to the top-level loop routine. Cached global variables are decached before returning.

The top-level loop dispatches to the function of the module for the topmost goal. Caching of global variables is done by passing them as parameters to functions implementing KL1 modules. By this, the caching code can be shared among modules.

Unification is expanded inline only for cases when one of the arguments is known to be an instantiated value (checked during compilation) and the other is a simple uninstantiated variable (checked dynamically). All other cases are treated by the runtime system.

3.6 Tracer

KLIC provides a simple stepping tracer. As will be described in section 5, recompilation takes considerably longer than conventional Prolog system. Thus, we designed a mechanism that does not require recompilation.

C functions corresponding to KL1 modules pop up the topmost goal from the goal stack, reduce it to children goals, and push them back. This procedure is normally repeated. For tracing, the heap limit is modified before calling the function to break the repetition after single reduction. Children goals can be recognized by remembering the next-to-top entry and comparing it with the goal stack entries after the reduction.

For children goals with different priority levels and goals waken up by variable instantiation cannot be found this way. Such goals are activated not by the compiled code but by the runtime subroutines. Thus, a tracing version of the runtime system is provided separately Only linking the program with this tracing runtime system will enable tracing. Relinkage is much less costly than recompliation.

Inline-expanded code (unification and arithmetics) cannot be traced this way. However, through our experiences, displaying the parent goal before and after reductions is enough.

4 Generic Objects

The generic object feature allows easy modification and extension of the system without changing the implementation core. This section briefly describes why and how such a feature is incorporated and how they are used.[8]

[8]We borrowed the idea of generic objects from AGENTS [7], modified and extended it for KLIC.

4.1 Objectives

With the KL1 implementations on PIM [3], we experienced severe difficulties in trying out different parallel execution schemes, as the schemes were too much integrated into the system core. This pointed to us a moral that system extensibility and modifiability should be put above bare efficiency. We needed a framework for such extension and modification.

On the other hand, as the system is for stock hardware, only limited number of tag bits can be handled efficiently. We thus needed some other ways to distinguish various built-in data types.

Generic objects were introduced to KLIC to achieve these two objectives at a time. Some of the built-in data types and non-local data references for parallel implementations are implemented as generic objects.

The core runtime system and compiled code only know that there are data types generically called "generic objects." Generic objects of all classes have the same interface; new object classes can be freely added without changing the system core.

4.2 Three Kinds of Generic Objects

Three kinds of generic objects are in KLIC.

Data Objects Immutable objects *without* time-dependent states. They are accessed via built-in predicates and generic method calls. Method calls will have an interface such as: `generic:Method(Obj, Args,...)`. Time-independence only means that they always look the same from KL1 programs; physical representations may be modified. For example, multiversion arrays are data objects, but their actual representation are mutated on updates.

Character strings and vectors are implemented as data objects. Various other objects, such as bignums are planned to be data objects.

Consumer Objects Mutable objects *with* time-dependent states. They look like logical variables with a goal awaiting for their instantiation. They are activated by variable instantiation, such as `Obj = [Method(Args, ...)|NewObj]`. Stream mergers are implemented as consumer objects.

Generator Objects Also mutable objects *with* time-dependent states. Unlike consumer objects, generator objects are activated by dereference operation.

Generator objects define a "generate" method. When a goal reduction requires the value of a variable associated with a generator object, the "generate" method of the object will be invoked. The object may generate some value immediately, or it may spawn a goal that will eventually instantiate the associated variable.

Generator objects were first introduced for nonuniform memory access. For example, in distributed memory implementations [9], remote memory references are represented as generator objects. In shared memory implementations also with local memory [8], they are used to access shared variables. On suspension of a goal, the "generate" method of a generator object associated with a suspension reason variable is called. It may send a value fetching message to a remote processor or read data from shared memory with required locking.

Generator objects can also implement lazy computation. As a generator object, generator goals can be activated from within the "generate" method, that is, only when the value of a variable is required by some other goals.

4.3 Data Representation

For all kinds of generic objects, their first word points to a method table, and the rest is defined only through their methods. Common operations such as copying for garbage collection are standard methods every object class has. Non-standard methods are called through the method "generic".

5 Performance Evaluation

Evaluation of the sequential core of KLIC is made through comparisons with similar language processing systems.

5.1 Conditions

We compared KLIC with two representative Prolog systems, **SICStus Prolog** version 2.1 patchlevel 8 [1] and **Aquarius Prolog** version 1.0 [6]. We also compared it with a Janus [10] to C compiler **JC** version 2.0 [4].[9] For SICStus, both native code (`fastcode`) and abstract machine code (`compactcode`) were measured.

The following benchmark programs are used for the comparison.

nrev: Naively reversing a list of 30 elements [13], repeated 10,000 times.

qsort: Quicksort of a list of 50 elements [13], repeated 10,000 times.

deriv: Symbolic differentiation [13]. Four runs with different data, times10, divide10, log10 and opt8, all repeated 100,000 times.

primes: Prime number generator using the sieve method. Generates and counts all prime numbers less than 10,000.

tak: Takeuchi's function, which makes many shallow recursive calls with simple integer arithmetics and comparisons. Called with arguments 24, 16 and 8.

All the measurements are on SparcStation 10 model 30, with 36 MHz SuperSparc processor without external cache, running SunOS 4.1.3. GCC 2.5.7 with options `-O2` and `-fomit-frame-pointer` is used for C compilation.

For KLIC, repetitions are through recursions. For Prolog systems, repetitions are made through backtracking as given in [13], as this seems to be the fastest. For JC, repetitions are by the measurement feature built into the system.[10] Thus, the timings for Prolog systems and JC do not include garbage collection while KLIC timings do.[11] Prolog timings are user time only while KLIC and JC timings include system time, but system time is not significant for these small benchmark programs anyway.

For the KLIC system, code sizes are measured using the Unix command `size` on relocatable objects generated from the KL1 programs. Thus, it does not include the runtime system. For about SICStus Prolog, code sizes are those reported by the compiler after the second compilation of the same file.

5.2 Results

Execution Speed Execution time for the benchmark programs are shown in table 1.

[9] A little further optimization were added to the released version 2.0.

[10] As described in appendix B.1 of the manual [5].

[11] GC overhead is not negligible. However, KLIC relies more heavily on GC than other systems, we thought including GC for KLIC should make comparisons fairer.

Table 1: Comparison of Execution Time

program	K	Sf	/K	Sc	/K	A	/K	J	/K
nrev	2,530	4,789	1.89	10,440	4.13	3,800	1.50	2,740	1.08
qsort	3,120	7,320	2.35	14,980	4.80	4,050	1.30	5,760	1.85
times10	2,540	5,659	2.23	12,569	4.95	5,120	2.02	3,260	1.28
divide10	3,000	5,890	1.96	14,390	4.80	5,890	1.96	4,370	1.47
log10	1,110	3,319	2.99	6,299	5.67	3,210	2.89	1,670	1.50
ops8	1,720	4,460	2.59	9,270	5.39	3,810	2.22	2,270	1.32
primes	1,550	2,869	1.85	5,349	3.45	3,200	2.06	2,130	1.37
tak	3,200	6,789	2.12	14,820	4.63	4,200	1.31	4,790	1.67
mean	2,210	4,908	2.22	10,343	4.68	4,077	1.84	3,108	1.41

K: KLIC; Sf: SICStus `fastcode`; Sc: SICStus `compactcode`; A: Aquarius; J: JC
Timing data are in milliseconds.
The "mean" row shows geometric means.

KLIC runs 1.3 to 3 times as fast as native code Prolog systems and 3.5 to 5.7 times as fast as abstract machine code of SICStus. JC is a little slower than KLIC but faster than Prolog systems. This seems reasonable as both JC and KLIC implement similar languages with similar schemes.[12]

Ralph Clarke Haygood took the effort of investigating the code generated by an earlier version of KLIC and his own improved version of SICStus[13] for `nrev` and `qsort`.

The nrev program runs about 50% slower than KLIC in his improved SICStus. For the list concatenation loop that dominates the performance, KLIC runs 20 instructions while the improved SICStus runs 32 per repetition. Out of the extra 12, 6 or 7 are due to the difference of the language: as KL1 does not have backtracking, unification is much simpler and thus is expanded in line. The rest, 5 or 6, may be got rid of by further improvements of SICStus, but that may need some non-trivial system redesign.

For about `qsort`, the dominating loop is that of partitioning a list into two, comparing elements with a pivot value. In typical cases, the KLIC runs 30 instructions per repetition while the improved SICStus needs 71. 27 outof the extra 41 are for choice point. Some of the remaining 14 may be optimized away but with substantial change on the system. The rest may be due to other language differences.

We have not investigated other programs in depth, but our conjecture is that about half of the speed advantage of KLIC comes from the language difference and the rest is from more elaborate system design. Low level optimizations, such as register allocation and instruction ordering, are by the C compiler in our case, while SICStus does it all by itself. That allowed us to concentrate better on higher level design issues, which seem to be more effective than precise control of lower level optimization.

Object Code Size Object code sizes are shown in table 2. Only SICStus Prolog was compared as separating program-specific and common codes was difficult for other systems.

As shown in the table, code sizes of KLIC are not so large. They are larger than abstract machine code but only by 25%, and smaller by about 30% than native code generated by SICStus.

[12]JC now does not have the two-occurrence restriction on variables, that made the language very close to KL1. JC compiles the whole program into one C function, while KLIC can compile module by module, but in these small benchmark programs, not many intermodule calls are made anyway.

[13]Not the released version that we used in the measurement.

Table 2: Comparison of Object Code Size

program	K	Sf	/K	Sc	/K
nrev	600	896	1.49	496	.827
qsort	976	1,216	1.25	672	.689
deriv	1,688	2,928	1.73	2,064	.818
primes	1,624	2,352	1.45	1,424	.877
tak	544	608	1.12	288	.529
mean	973	1,354	1.39	776	.798

K: KLIC; Sf: SICStus `fastcode`; Sc: SICStus `compactcode`
Code sizes are in bytes.
The "mean" row shows geometric means.

Compilation Time One problem with KLIC is its slow compilation speed. While SICStus can compile each program within 100 milliseconds, KLIC requires a few seconds. This is similar for JC. The compilation time is dominated by the C compiler for both KLIC and JC. Although KLIC is much faster than Aquarius that requires more than 20 seconds even for `nrev`, recompilation has to be avoided as far as possible. This led us to a design with separate compilation and the tracing scheme without recompilation.

6 Conclusion

A scheme for portable implementation of KL1 is described. The scheme employs the strategy to compile KL1 source code into C for increased portability, rather than directly generating machine code. The sequential core its experimental implementation shows that efficiency in both time and space aspects can be achieved even by such an indirect code generation scheme.

Various efforts are on-going to make the system more useful in parallel software research, including the following.

- Implementation of more language features, such as floating points
- Further optimization through static analyses
- Both shared- and distributed-memory parallel implementations
- Providing better software development tools, such as tuning tools

7 Acknowledgments

Hiroshi Yashiro contributed in early design phases of the system. Goichi Nakamura implemented the macro library for defining generic objects. Kazuaki Rokusawa, Akihiko Nakase and Masao Morita contributed to the system design through their investigation of parallel implementation.

Discussions with Saumya Debray and Koenraad De Bosschere were helpful in earlier design phase of the system. Ralph Clarke Haygood gave in-depth information of SICStus Prolog. David Gudeman and Peter A. Bigot helped us comparing KLIC with the JC system. Discussions with Kazunori Ueda, Hideo Kawai, Katsumi Takahashi, Hiroshi Nakashima and Kenji Horiuchi also helped the design.

References

[1] Mats Calsson, Johan Widén, Johan Andersson, Stefan Andersson, Kent Brootz, Hans Nilsson, and Thomas Sjöland. *SICStus Prolog User's Manual*, 1993.

[2] Takashi Chikayama, Hiroyuki Sato, and Toshihiko Miyazaki. Overview of the parallel inference machine operating system (PIMOS). In *Proceedings of FGCS'88*, pages 230–251, Tokyo, Japan, 1988.

[3] Atsuhiro Goto, Masatoshi Sato, Katsuto Nakajima, Kazuo Taki, and Akira Matsumoto. Overview of the parallel inference machine architecture (PIM). In *Proceedings of FGCS'88*, Tokyo, Japan, 1988.

[4] David Gudeman, Koen De Bosscherre, and Saumya K. Debray. `jc`: an efficient and portable sequential implementation of janus. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*. The MIT Press, 1992.

[5] David Gudeman and Sandra Miller. *User Manual for* `jc` *– A Janus compiler, version 2.0*, 1994.

[6] Rlaph Clarke Haygood. *Aquarius Prolog User Manual*, 1993.

[7] Sverker Janson, Johan Montelius, Kent Boortz, Per Brand, Björn Carlson, Ralph Clarke Haygood, Björn Danielsson, and Seif Haridi. AGENTS user manual. SICS technical report, Swedish Institute of Computer Science, 1994.

[8] Masao Morita, Nobuyuki Ichiyoshi, and Takashi Chikayama. A shared-memory parallel execution scheme of KLIC. ICOT technical report, ICOT. *to appear.*

[9] Kazuaki Rokusawa, Akihiko Nakase, and Takashi Chikayama. Distributed memory implementation of KLIC. ICOT technical report, ICOT. *to appear.*

[10] Vijay A. Saraswat, Ken Kahn, and Jacov Levy. Janus: A step towards distributed constraint programming. In *Proceedings of North American Conference on Logic Programming*. The MIT Press, October 1990.

[11] Yasutaka Takeda, Hiroshi Nakashima, Kanae Masuda, Takashi Chikayama, and Kazuo Taki. A load balancing mechanism for large scale multiprocessor systems and its implementation. In *Proceedings of FGCS'88*, Tokyo, Japan, 1988. Also in New Generation Computing 7–2, 3 (1990), pp. 179–195.

[12] Kazunori Ueda and Takashi Chikayama. Design of the kernel language for the parallel inference machine. *The Computer Journal*, December 1990.

[13] David H. D. Warren. Implementing Prolog – compiling predicate logic programs. Research Reports 39 and 40, Department of Artificial Intelligence, University of Edinburgh, 1977.

[14] David H. D. Warren. An abstract Prolog instruction set. Technical Note 309, SRI International, 1983.

Andorra-I Compilation

Vítor Santos Costa,[†]**Inês Dutra**, and **Rong Yang**[‡]

Abstract

Andorra-I is an experimental parallel Prolog system that transparently exploits both dependent and-parallelism and or-parallelism. One of the main components of Andorra-I is its preprocessor. The preprocessor supports Prolog applications through a sequencer, that guarantees correct execution of Prolog features such as side-effects and cuts. The sequencer is assisted by an abstract interpreter to detect cuts and meta-predicates that need sequencing. A determinacy analyser generates routines that are used to verify at run-time when goals are determinate. Finally, a clause compiler generates WAM code for each clause. We describe the components of the preprocessor and present recent work on improving Andorra-I performance through better compilation.

1 Introduction

Andorra-I [7, 14] is a parallel logic programming system that supports both and- and or-parallelism. Experience in using Andorra-I has led to the following main conclusions:

- Andorra-I performs well and exploits parallelism successfully for Prolog applications, committed-choice style applications, and new, "Andorra style", applications.
- The coroutining in the Basic Andorra Model can reduce the search space of logic programs. This holds true for some Prolog programs, and has been exploited in the "Andorra style" applications.

The Andorra-I preprocessor [9, 6] is one of the key components of the Andorra-I system. The preprocessor was designed to obtain correct and efficient execution of Prolog programs in Andorra-I. To allow the correct execution of Prolog programs, we researched the operation of Prolog programs with traditional left-to-right selection function, and investigated which features allow early execution of goals, and for which features left-to-right needs to be enforced. To obtain efficient execution of programs in Andorra-I, we designed a compiler for Andorra-I. The compiler had to address the new characteristics of Andorra-I's selection function, and particularly the problem of determinacy detection. Therefore, the main innovation of the preprocessor over previous compile-time analysis tools was addressing two new problems in Andorra-I: which goals *can* be executed early, and which goals *must not* be executed early.

We found that a goal must not be executed early if it interferes with the correct operation of some builtin, such as a side-effect predicate or cut. This is rather hard to detect at run-time. We use the principle that some calls in the program are "sensitive",

[†]Laboratório de Inteligência Artificial e Ciências da Computação, Universidade do Porto, Portugal
[‡]Dept. of Computer Science, University of Bristol, Bristol, UK

i.e., may behave incorrectly if goals executed later in the left-to-right execution are executed early. Compile-time analysis detects such calls, and detects for which goals it should restrict early execution.

In Andorra-I a goal can be executed early if it is determinate. Andorra-I generates at compile-time some code for each procedure. This code must detect, very quickly at run-time, whether a call is determinate.

The tools we described were integrated as a *sequencer*, that generates code to prevent early execution of goals, and a *determinacy code analyser*, that compiles code to detect determinate goals. The code generated by the preprocessor is executed by the Andorra-I *engine*.

The sequencer benefits from global information generated by an *abstract interpreter*. The information is mainly useful in detecting uses of cut and of meta-predicates that are not sensitive, and thus prevents some unnecessary sequencing [8].

The determinacy code analyser was originally designed to compile only determinacy code, but was since extended to fully compile the (Andorra-I) Prolog source program. A *clause compiler* generates code for individual clauses. The determinacy compiler integrates this code with the determinacy code to obtain the full procedure code. The end-code can then be run by the Andorra-I engine.

Andorra-I brings important advantages to logic programming. As always, there is a cost, particularly, some performance overhead when comparing to pure sequential Prolog, or-parallel systems, or recent implementations of the committed-choice languages. In this work we concentrate on how Andorra-I performance can be improved up to levels closer to these systems. Several approaches are possible:

- Fine-tuning the current engine and compiler. The current system is not as well optimised as, say, current Prolog implementations, so improvements could be substantial.
- Moving from an abstract machine interpreter to a native language implementation. This has been tried for Prolog and for committed-choice languages, where it gives substantial improvements in performance.
- Using global analysis to reduce overheads.

Substantial improvements on Andorra-I performance will demand work both on the engine and preprocessor components. In this paper, we concentrate on the preprocessor component, and how compilation can be a useful technique to improve the current performance of Andorra-I.

The paper is organised as follows. First, we present the Andorra-I system, and discuss where performance gains may be obtained. Next, we review the components of the preprocessor, and analyse how they can be improved and adapted for our goals of better performance. Lastly, we present the structure of a new preprocessor and conclude the paper.

2 Andorra-I

Andorra-I is a logic programming system that implements the Basic Andorra Model. In this model:

- Goals can execute in and-parallel, provided they are *determinate*;

- If no (selectable) goals are determinate, we can select one non-determinate goal, and explore its alternatives, possibly in or-parallel.

The model provides both and-parallelism between determinate goals, and or-parallelism between alternatives for non-determinate goals. Or-parallelism can be exploited in the style of Aurora. And-parallelism can be exploited in the same vein as for the committed-choice languages.

Executing determinate goals first also gives a form of coroutining similar to the coroutining in the committed-choice languages. This allows Andorra-I to run programs originally written for the committed-choice languages. It also opens new applications, where this coroutining can be used to reduce the computation search space. In this vein we say that Andorra-I supports *Andorra-I Prolog*, a language that extends Prolog with the implicit coroutining available in the Andorra selection function.

Note that in Andorra-I coroutining arises from the determinacy property of goals. Behaviour similar to KL1 or committed-choice languages is therefore possible by using pruning operators for determinacy. Andorra-I supports cut and commit. In Andorra-I Prolog, goals with cuts can be determinate and the clause for a cut be taken if either one of the following conditions is satisfied:

1. There is a single matching clause, regardless of cuts; or,
2. All previous clauses cannot succeed, and head unification plus all goals to the left of the cut, can succeed without binding variables. Currently, Andorra-I only verifies whether builtin goals succeed.

Clauses with commits can be taken if either condition (1) holds, or head unification plus all goals to the left of the cut can succeed. Note that in Andorra-I a cut can be executed even if head unification is not read-only. This is somewhat different from traditional committed-choice languages.

Andorra-I is organised as preprocessor, that performs compile-time analysis, and the run-time environment, that supports the actual execution of programs. We next review these components and study their performance.

2.1 The Andorra-I Run-Time Environment

Andorra-I programs are executed by *teams* of abstract processing agents called *workers*. Each worker usually corresponds to a physical processor. Each team, when active, is associated with a separate or-branch in the computation tree and is in one of two computation phases:

Determinate For a team, as long as determinate goals exist in the or-branch, all such goals are candidates for immediate evaluation, and thus can be picked up by a worker. This phase ends when no determinate goals are available, or when a determinate goal fails. In the first case, the team moves to the non-determinate phase. In the second case, the corresponding or-branch must be abandoned, and the team will backtrack in order to find a new or-branch to explore.

Non-determinate If no determinate goals exist, the leftmost goal (or a particular goal specified by the user) is reduced. A choicepoint is created to represent the fact that the current or-branch has now forked into several or-branches, while the team itself will explore one of the or-branches. If other teams are available, they can be used to explore the remaining or-branches.

Most of the execution time in Andorra-I should be spent in the engine, executing programs. Workers will enter an *and-scheduler* to search for determinate goals, or to synchronise in case of failure or when no determinate goals are available. Teams will enter an *or-scheduler* [1] when they have no more alternatives to exploit, and therefore need to move in the search tree. Finally, teams may need to be reconfigured as the computation progresses, hence Andorra-I includes a *top-scheduler* to reconfigure teams whenever needed [4].

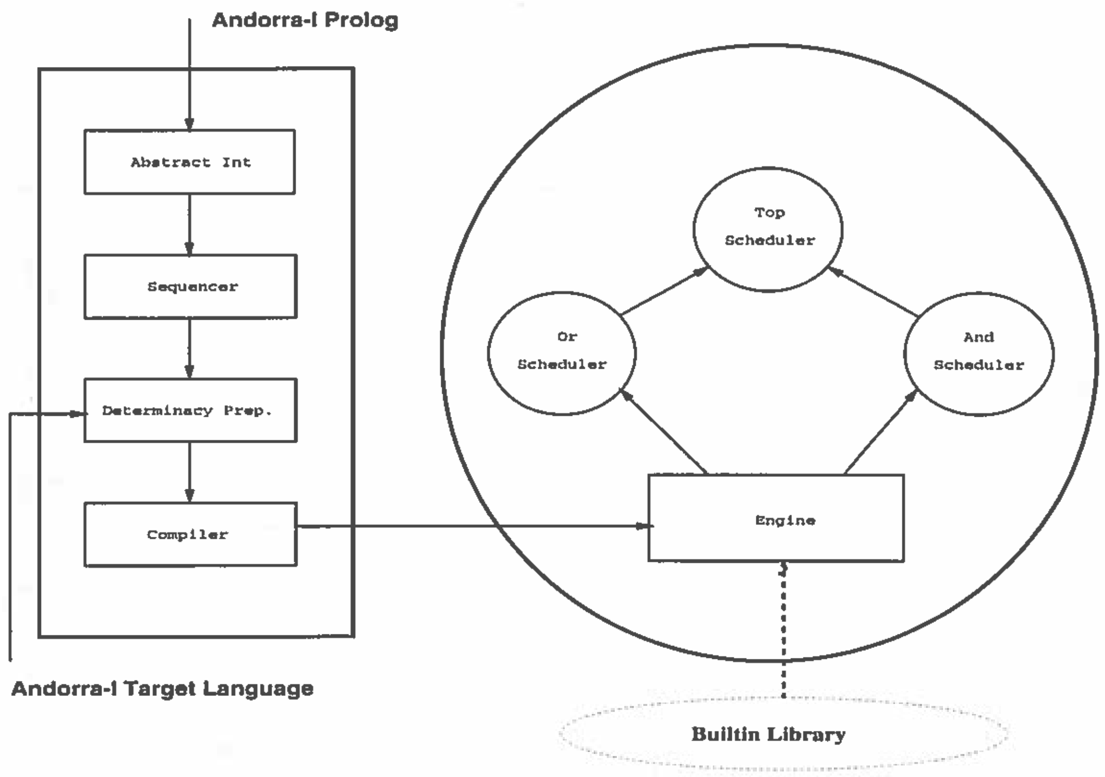


Figure 1: The Andorra-I Architecture

Figure 1 describes the full Andorra-I structure, including both the run-time and compile-time components of the system.

2.1.1 The VRAM

The initial version of Andorra-I was an interpreter. To obtain better performance a compiled version was developed. The compiled system uses a WAM-like abstract instruction set, which is then emulated by the engine. We call this abstract instruction set VRAM. The compiled system and its performance has been previously described [14]. We briefly mention the main characteristics:

• The development of this abstract machine was influenced by Crammond's abstract machine for Parallel PARLOG, JAM [3]. Data structures mainly follow from Prolog and JAM, although in relation to JAM there is the important difference that not all variables are globalised. Instructions are organised into the following groups, (i) unification instructions, (ii) argument writing instructions, (iii) head and guard matching instructions, (iv) goal management instructions, (v) choice-point management instructions, and (vi) determinacy testing instructions.

• Instructions groups (i) and (ii) are very similar to the corresponding WAM instructions [13]. The main novelty are changes to support locking of compound terms in write-mode. Andorra-I avoids excessive locking by only actually unifying the newly created structure with the variable when a term has been fully created.

• Instruction group (iii), head and guard matching instructions, is very similar to group (i), the main difference being that these instructions can only test arguments, but can never actually bind them, whereas instructions in group (i) can. Matching (or read-mode) instructions can be called to verify if a goal is determinate due to cut or commit.

• Instruction group (iv), goal manipulation instructions, is the most interesting one. Andorra-I is a pure goal-stacking system. Goal-frames in Andorra-I are reasonably complex. To minimise overheads, Andorra-I tries to reuse registers wherever possible. This is not possible if the current reduction was non-determinate, or if we are not expanding a goal on the top of the stack.

• Instruction group (v), choice-point manipulation instructions, follows the same principles as for the corresponding Prolog instructions.

• Instruction group (vi), determinacy instructions, are a generalisation of the Prolog indexing instructions. Determinacy instructions in Andorra-I can select on any arguments, can fetch arguments from the goal, and can call builtins whenever necessary.

In a traditional Prolog implementation, the most expensive instructions are the choice-point manipulation instructions, group v. Control instructions are not as expensive, but still quite frequent. Most unification instructions are cheap, but on the other hand they form the bulk of the code (the exception is general unification, that in some rare cases can be quite expensive). High-performance Prolog implementations [12, 10] bypass the WAM and generate low-level code. The ensuing optimisations are quite effective for unification instructions, but to obtain maximum performance these systems do also need to limit the number of choice-point operations.

Committed-choice implementations do not need to support group (v) instructions. In relation to Prolog, overheads are more evenly distributed. If the goal frame is complex call instructions may be more expensive than in Prolog. This is the case for Parlog, and also for Andorra-I, as in this case goal frames need to explicitly support goal ordering information, which is implicit in Prolog. Unification and test instructions, i.e., instructions (i), (ii), and (iii) should be as expensive as Prolog, except if test instructions force a goal to suspend, or if unification instructions awake a set of goals. In these cases the instructions can become quite expensive. Therefore, to obtain high performance in these languages one needs to attain:

- Low overhead in execution instruction, which can be attained by native code compilation [11, 2].
- Low suspension rate, which depends on goal scheduling.

Execution in Andorra-I is a combination of Prolog and of the committed-choice languages. Therefore one can generally expect similar instruction weights and execution patterns. Still, there are some novel aspects:

Determinacy Andorra-I searches for determinacy, and hence tends to create less choice-points than Prolog. Even when a goal is not determinate, Andorra-I can benefit because determinacy analysis may have excluded quite a few clauses, and the current implementation already passes this information to the non-determinate selection scheme.

The reverse of the medal is that Andorra-I may suspend more often than a committed-choice language. Usually, committed-choice procedures have a single way of committing to a clause. This contrasts with the observation that there may exist many ways of making a goal determinate. For instance, consider the well-known procedure append:

```
append([], L, L).
append([H|L1], L2, [H|NL]) :- append(L1, L2, NL).
```

The goal is determinate if the first argument is bound, if the third argument is bound to something other than a list, or if the second and third arguments are bound to different values. In contrast, in a language such as KL1 there would be a single way to use this procedure. Most often, it would be used as:

```
append([], L, OL) :- OL = L.
append([H|L1], L2, OL) :- OL = [H|NL] | append(L1, L2, NL).
```

Only the first argument would make the goal determinate, and the goal will suspend at most in a single argument. If the compiler knows a goal will suspend on a single variable, goal suspension can be optimised [3].

If one looks at real programs, experience has shown that there are two main types of calls:

- Calls to recursive goals may be determinate if their expected input arguments are bound, or as in the example, may sometimes become determinate if their expected output arguments is bound. Usually the second case is "bogus", that is, it rarely happens and if it will happen it will not improve performance..
- The separation between input and output is not so clear for non-recursive procedures. Users tend not be as strict concerning execution of non-recursive goals, and if there is a way to make a call to a non-recursive procedure determinate, there may be a point in trying so.

We can gain the most from reducing determinacy overheads in the recursive goals. This links to the problem of "optimal scheduling" for the committed-choice languages: to know which goals have the bindings, we need to know where the bindings are!

Team Synchronisation As a parallel system, Andorra-I brings the novel issue of worker synchronisation within a team. This problem is particularly important in applications which have combined parallelism, that is, where there are determinate sequences of executions intermixed with the creation of goals. In contrast to the committed choice languages where deadlock and failure are rather unfrequent operations, and and-scheduling is optimised for the standard case of work-search, Andorra-I needs to support an efficient acknowledgement of choice-point creation and of backtracking.

A related issue is non-determinate goal selection. Andorra-I selects the leftmost goal to emulate Prolog. Even if Prolog implementation was not supported, the system would still need to very quickly find a non-determinate goal to execute. Thus, Andorra-I must quickly perform:

- Team Synchronisation, in the case of team deadlock and failure or backtracking
- Goal Search, in the case of team deadlock.

Currently, the engine and and-scheduler address this issue with little aid from the compiler. A synchronisation protocol was designed to handle communication between workers in a team. Goals are connected within a data structure, the sideways-linked chain. In order to obtain maximum performance, we would like to go further, and use specific information from actual programs to reduce these overheads.

Memory Management Memory usage is an important issue in program development. This is particularly true for large applications, where memory performance may make the actual difference between running or not running the application.

In this regard, Prolog and the committed choice languages have very different behaviour. Execution in the committed-choice languages will allocate space which is mainly recovered through garbage-collection (schemes for dynamic recovery of space have been proposed, though). In Prolog, the need for garbage collection is lesser, mainly due to backtracking, but also thanks to the design of Prolog data-structures.

Memory management in Andorra-I combines the issues in Prolog with the issues in the committed-choice languages. Programs which incur in much backtracking will recover space easily. On the other hand, large committed-choice applications which do never backtrack still demand garbage collection to be runnable in Andorra-I.

3 The Preprocessor

We next describe the main components of the preprocessor. The preprocessor was designed to allow correct and efficient program execution in Andorra-I. In order to simplify the design and maintenance of the system, the three components for the preprocessor are organised as follows:

- The *abstract interpreter* receives a Prolog (or more precisely, Andorra-I Prolog) program. It analyses that program according to the Prolog selection function, and adds mode information.

- The *sequencer* receives the annotated program. It verifies whether all conjunctions can be run in parallel, or whether some need to be run sequentially to obtain Prolog compatibility. If so, it adds the corresponding declarations.

- The *determinacy analyser* receives the final transformed program, generates determinacy code, and calls the *compiler* to obtain compiled code for each clause.

The preprocessor is organised in such a way that any component will not depend on the previous components. If a user does not want to worry about sequencing, the user can call the determinacy analyser directly. Also, the user may bypass the abstract interpreter by entering input mode declarations.

The two first modules perform global analysis of the program. Whereas most global analysis systems are designed to improve performance, our sequencer and abstract interpreter are mainly designed to guarantee correct execution of Prolog programs. In the next sections we will be interested on how the information they provide can also be used to improve execution efficiency.

We next discuss the preprocessor components in more detail.

3.1 The Abstract Interpreter

Our motivation in designing the abstract interpreter is to allow parallelism near builtins such as `cut`, `commit`, and `atom`. In most cases these builtins will allow early execution of goals nearby, but especially in the case where Prolog calls them with uninstantiated arguments, early execution of other goals may prevent their correct execution. In general, we call such goals *sensitive*, Our main observation was that:

If builtin execution was sufficiently instantiated in Prolog, we can guarantee it will also be sufficiently instantiated in Andorra-I.

Therefore, we can find all the builtins that were sufficiently instantiated in Prolog, and allow early execution of goals near these builtins.

Our motivation in designing the abstract interpreter was to use this principle to detect which goals are sensitive without programmer input. The machinery of the abstract interpreter has been described in [6]. Its main characteristics are:

- The abstract domain is a level k domain with a recursive data structure, the list.
- Information can be stored for every call, or for every procedure. The interpreter allows a limited amount of automatic specialisation.
- With the current fixed point calculation algorithm, the abstract interpreter is about four times slower than the SICStus compiler. This varies wildly, and depends on the program and on the parameters we give to abstract interpreter.

We have analysed the performance of the abstract interpreter. Our main conclusion was that the system could perform quite well for some applications, but not so well for some applications. In general, abstract interpretation has two main problems:

- It is quite difficult to reason on the state of partial data structures, such as open lists or open binary trees. Depth-k domains are not really a solution for this problem (note that sharing between these structures may be easier to recognise).
- Programmers may sometimes break discipline. For instance, if arguments are unnecessary, it is common to leave unbound variables. This can be propagated throughout the program tree.

This means that abstract interpretation is usually not very "robust" for real applications. In [6], we presented some suggestions on improvements for the abstract interpreter. In this paper, we shall concentrate on the other issue of how abstract interpretation can be used for improving Andorra-I performance.

What are the advantages of abstract interpretation? Our previous discussion should have made it clear where we can benefit the most from abstract interpretation:

Input and Output arguments for recursive goals. By using mode information we can detect which arguments are input for recursive goals. This allows some simplification of the determinacy code, and in general would result in better compiled code.

Producer-Consumer relationships between goals, or more precisely, detecting whether a goal generates bindings that make other goals determinate. This gives an ordering for Andorra-I execution, which is important to detect grain-size for and-tasks, can be used for compile-time scheduling and for aiding run-time scheduling. Moreover, if we know the computation ordering we may be able to use some Prolog optimisations,

Prolog Abstract Interpretation To know the actual program execution patterns we would need to perform abstract interpretation based on the Andorra selection function. This for instance the case for committed-choice language programs or new, Andorra style, applications. On the other hand, for Prolog applications, we can try to take advantage of our current system. We have previously found that the following relations hold:

- Goals that were determinate when executed by Prolog will also be determinate when executed by Andorra-I.
- Goals that were non-determinate in Prolog may be determinate in Andorra-I.

In general, this means that the flow of control and data may be different in Prolog and Andorra-I. Prolog based abstract interpretation will therefore give us only a first approximation to the actual Andorra-I operational semantics. Still, this approximation can be quite useful. E.g., if we detect that goals in a program are always non-determinate in Prolog and disallow and-parallelism, we can benefit by not supporting and-parallelism at all.

3.2 The Sequencer

The sequencer was designed to:

- Detect which goals are sensitive in the Prolog execution.
- Generate annotations (sequential conjunctions) to prevent early execution of goals that can interfere with sensitive goals.

The sequencer performs global analysis of the program. It analyses cuts, meta-predicates and side-effects locally. It then uses a simple global propagation algorithm to detect which calls should be sequenced, and generates the necessary sequential conjunctions.

3.3 The Determinacy Analyser and the Compiler

The determinacy analyser and the compiler work together to generate code for every procedure. The two components perform local analysis for each procedure, and generate the code according to the following principles:

- The determinacy compiler looks at the heads of the clauses, and at builtins in the body, to detect sufficient conditions for goal determinacy. These conditions are then translated into "decision tree"-like code.
- The compiler looks individually at each clause, and generates VRAM instructions. The compiler applies a direct mapping from symbols in the clauses to VRAM instructions, followed by some amount of register optimisation.

The determinacy analyser has been developed as a compromise between our original goal of detecting *all* determinate goals, and efficiency. In practice, this compromise has been shown to work quite well. Still, there are some important limitations to the system. The most important is *determinacy through user-defined goals.*

In general the only way of detecting whether at most a clause will succeed for a goal is by solving the goal! Still, it is often the case in Prolog that determinacy cannot be found from the builtins in a procedure, but could have been easily found if goals in the body were studied. Two typical examples in Prolog are:

```
compare(X, Y, X) :- greater(X, Y).
compare(X, Y, Y) :- smalleq(X, Y).

intersection([], _, []).
intersection([H|Ls], L, [H|NLs]) :- member(H, L), !,
        intersection(Ls, L, NLs).
intersection([_|Ls], L, NLs) :-
        intersection(Ls, L, NLs).
```

In the first case, **greater** could be a simple call to `X < Y`. In the second example, if the first two arguments are instantiated **member/2** could be executed by the determinacy code.

We have proposed two solutions for these problems:

- Extending the current preprocessor algorithm to performed a limited amount of non-local analysis, and extend the VRAM to allow entering user-defined goals in the determinacy code.

- Use program transformation to process the previous programs into simpler programs where determinacy is easier to detect. In the example, unfolding and backtracking removal could be applied to the programs:

  ```
  compare(X, Y, X) :- X > Y.
  compare(X, Y, Y) :- X =< Y.

  intersection([], _, []).
  intersection([H|Ls], L, OLs) :-
          member(H, L, Yes),
          output_on_yes(H, NLs, OLs),
          intersection(Ls, L, NLs).
  ```

  Note that the transformations themselves are straightforward. The main problem will be *when to apply them.*

Comparing the two approaches, the main advantage of the first one is that we preserve the structure of the original program. Still, we believe the second approach should be followed, mainly because one should keep the abstract machine instruction set as simple as possible. Having fewer instructions simplifies the abstract engine emulator, makes it easier to generate correct abstract machines, and will allow an easier transition to a native code implementation.

4 Reorganising the Preprocessor

To obtain a high-performance Andorra-I implementation, it is desirable to (a) specialise the cases where execution simplifies to a Prolog-like or Committed-Choice Language execution only, (b) extend the determinacy detection algorithm to detect non-flat cases of determinacy, thus removing unnecessary bottlenecks in the and-parallelism, and (c) improve efficiency and execution speed at the instruction level. Figure 2 presents a new organisation for the preprocessor that is designed to support these goals.

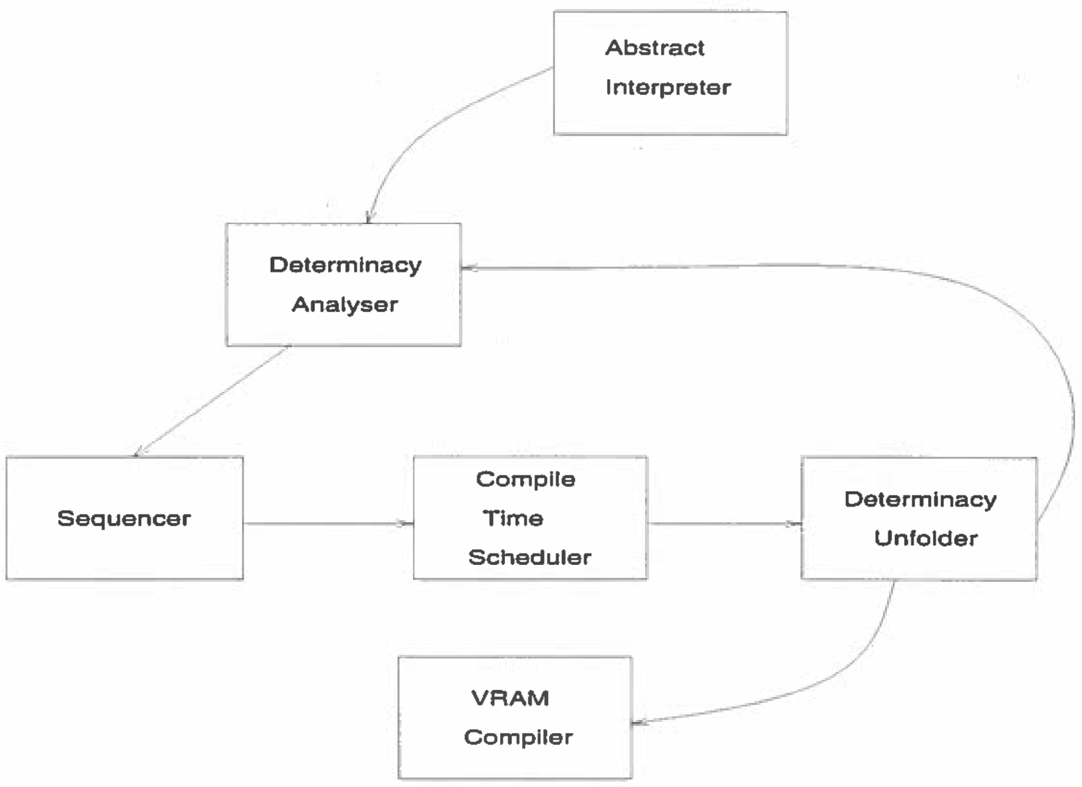


Figure 2: A New Preprocessor

In relation to our original approach, shown in figure 1, the new entities will be the determinacy unfolder, discussed in section 3.3, and the compile-time scheduler, whose need was discussed in section 2.1. The current determinacy analyser will be split into two components. The first finds the conditions necessary to make a goal determinate. Based on the determinacy information, plus mode information from the abstract interpreter or from the user, the sequencer, compile-time scheduler, and determinacy unfolder will transform the program, by rewriting procedures that can be made flat determinate and adding pragmas to be interpreted by the compiler, or at run-time by schedulers.

For hystorical reasons the preprocessor has separated generation of code for the decision tree from generation of code for the clauses. In the new scheme, both code generation for the determinacy tree and clause compilation will be integrated together, which should allow better register optimisation and code reuse.

5 Conclusions

The usefulness of a programming language is mainly a function of two factors: performance and expressiveness. We believe that Andorra-I provides a very expressive form of control, which has been proven very useful both for logic programming and constraint logic programming applications [5]. In terms of performance, Andorra-I has obtained results that are comparable to standard logic programming applications, while being the first system we know that exploits both dependent and- and or-parallelism.

Still, and as logic programming implementation technology moves on, techniques such as native code compilation, program specialisation, and compile-time scheduling show promise for even better performance of logic programming systems. We discussed how these techniques can be applied to Andorra-I, and presented a framework that will integrate these techniques into a new version of the Andorra-I preprocessor.

References

[1] A. Beaumont, S. M. Raman, P. Szeredi, and D. H. D. Warren. Flexible Scheduling of OR-Parallelism in Aurora: The Bristol Scheduler. In *PARLE91: Conference on Parallel Architectures and Languages Europe*, volume 2, pages 403–420. Springer Verlag, June 1991.

[2] T. Chikayama, T. Fujise, and H. Yashiro. A portable and reasonably efficient implementation of KL1. In D. S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming*, page 833, Budapest, Hungary, 1993. The MIT Press.

[3] J. A. Crammond. The Abstract Machine and Implementation of Parallel Prolog. Technical report, Dept. of Computing, Imperial College, London, June 1990.

[4] I. Dutra. A Flexible Scheduler for the Andorra-I System. In *LNCS 569, ICLP'91 Pre-Conference Workshop on Parallel Execution of Logic Programs*, pages 70–82. Springer-Verlag, June 1991.

[5] S. Gregory and R. Yang. Parallel Constraint Solving in Andorra-I. In *International Conference on Fifth Generation Computer Systems 1992*, pages 843–850. ICOT, Tokyo, Japan, June 1992.

[6] V. Santos Costa. *Compile-Time Analysis for the Parallel Execution of Logic Programs in Andorra-I.* PhD thesis, University of Bristol, August 1993.

[7] V. Santos Costa, D. H. D. Warren, and R. Yang. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-Parallelism. In *Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming PPOPP*, pages 83–93. ACM press, April 1991. SIGPLAN Notices vol 26(7), July 1991.

[8] V. Santos Costa, D. H. D. Warren, and R. Yang. The Andorra-I Engine: A parallel implementation of the Basic Andorra model. In *Proceedings of the Eighth International Conference on Logic Programming*, pages 825–839. MIT Press, June 1991.

[9] V. Santos Costa, D. H. D. Warren, and R. Yang. The Andorra-I Preprocessor: Supporting full Prolog on the Basic Andorra model. In *Proceedings of the Eighth International Conference on Logic Programming*, pages 443–456. MIT Press, June 1991.

[10] A. Taylor. LIPS on a MIPS: Results from a Prolog Compiler for a RISC. In *Proceedings of the Seventh International Conference on Logic Programming*, pages 174–185. MIT Press, June 1990.

[11] E. Tick and C. Banerjee. Performance evaluation of Monaco compiler and runtime kernel. In *ICLP93*, pages 757–773, 1993.

[12] P. Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming?* PhD thesis, University of California at Berkeley, November 1990.

[13] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, 1983.

[14] R. Yang, T. Beaumont, I. Dutra, V. Santos Costa, and D. H. D. Warren. Performance of the Compiler-based Andorra-I System. In *Proceedings of the Tenth International Conference on Logic Programming*, pages 150–166. MIT Press, June 1993.

Shared Paged Binding Array: A Universal Datastructure for Parallel Logic Programming

Gopal Gupta
Enrico Pontelli

Lab for Logic and Databases
Dept of Computer Science
New Mexico State University
Las Cruces NM USA
gupta,epontell@nmsu.edu

Vítor Santos Costa

Advanced Computing Research Center
Dept of Computer Science
University of Bristol
Bristol BS8 1TR UK
costa@compsci.bristol.ac.uk

Abstract

Two major problems that arise in parallel logic programming systems are: (i) redundant computation during and-parallel execution of dependent goals, and, (ii) efficient representation of multiple environments at runtime. Both these problems are caused by non-determinism present in logic programs—responsible for much of the power of logic programming. This paper is mainly concerned with solving the second problem, namely, the efficient representation of multiple environments at runtime in parallel logic programming systems. We present a datastructure called the *Shared Paged Binding Array* that arguably is most suited for implementing any arbitrary parallel logic programming system (i.e., a system that exploits any arbitrary combination of dependent-and parallelism, independent and-parallelism and or-parallelism). This datastructure can also be used for and-or parallel execution of Committed Choice Languages with Deep Guards as well as for realizing the implementations of more advanced models of parallel execution such as the Extended Andorra Model and the Andorra Kernel Language. Details of the Shared Paged Binding Array are presented, its merits over other techniques are shown, and, finally, implementations that combine different flavors of and- and or-parallelism using Shared Paged Binding Arrays are briefly sketched.

1 Introduction

Logic Programming is a declarative programming paradigm that is becoming increasingly popular. One of the distinguishing features of logic programming languages is that they allow considerable freedom in the way programs are executed. This latitude permits one to exploit parallelism implicitly (without the need for programmer intervention) during program execution. Indeed, three main types of parallelism have been identified and successfully exploited in logic programs:

(i). *Or-parallelism:* (abbreviated OP) arises when more than a single rule define some relation and a procedure call unifies with more than one rule head—the corresponding bodies can then be executed in or-parallel fashion. Or-parallelism is thus a way of efficiently searching for solution(s) to the top-level query.

(ii). *Independent And-parallelism:* (abbreviated IAP) arises when more than one goal is present in the query or in the body of a procedure, and the run-time bindings for the variables in these goals are such that two or more goals are *independent* of one another, that is, their resulting argument terms after applying the bindings of the variables are either variable-free (i.e., *ground*) or have non-intersecting sets of variables [22].

(iii). *Dependent And-parallelism:* (abbreviated DAP) arises when mutually dependent goals that share a common variable are executed in parallel to co-operatively produce a binding for this variable. Dependent and-parallelism is readily found in applications that involve producer-consumer and reactive interactions [27].

Parallel Execution of Logic Programs has been an active area of research since a decade [24, 5, 7, 26]. However, parallel logic programming systems that can *efficiently exploit all* forms of parallelism present in Logic Programs—or-parallelism, independent and-parallelism, and dependent and-parallelism—still elude us. The main source of difficulty has been the non-determinism—responsible for much of the power of logic programming—present in logic programs. Non-determinism affects parallel execution in two ways: (i) Unrestricted parallel execution of non-determinate goals that have data-dependencies between them leads to redundant computations being performed; (ii) Due to non-determinism, variables in goals may be imparted multiple bindings during or-parallel execution, since each matching clause may bind the variable differently; efficient maintenance and representation of these multiple environments is a non-trivial problem which is compounded further in the presence of independent and dependent and-parallelism. We believe that problem (i) demands appropriate control mechanisms in logic programming languages, while problem (ii) can be solved by designing smart datastructures which allow for efficient representation and maintenance of multiple environments. This paper is concerned with solving problem (ii). We present a datastructure called *Shared Paged Binding Array*, and show how it can be (universally) used for implementing any *arbitrary* parallel logic programming system, i.e., a system that exploits any arbitrary combination of OP, (determinate or non-determinate) DAP and (determinate or non-determinate) IAP. In particular, we will show how the Shared Paged Binding Array datastructure can be used for supporting efficient implementation of systems combining:

1. OP and IAP (with goal recomputation)

2. OP and IAP (with goal reuse)

3. OP and determinate DAP

4. OP, determinate DAP, and IAP

5. OP, non-determinate DAP and IAP

6. Committed Choice Languages with deep guards

It can also be used for implementing parallel Constraint Logic Programming Systems.

The rest of the paper is organized as follows: In Section 2 we further elaborate on problems related to control in dependent and-parallel execution. In Section 3 we discuss the problem of representing multiple environments that arise due to or-parallelism in parallel logic programming systems. We briefly discuss criteria, based on very pragmatic considerations, that any solution to this problem should satisfy. We mention the various earlier attempts and describe how the lack of a general and efficient environment representation technique has hindered progress in parallel logic programming. In Section 4 we describe the Binding Arrays technique that has been used for implementing purely or-parallel systems and show why it is ineffective when and-parallelism is present as well. In Section 5 we present an extension of Binding Arrays called Shared Paged Binding Arrays that satisfies all the criteria developed in earlier sections. We next show in Section 6 how different systems that combine different forms of parallelism (listed above) can be realized using the Shared Paged Binding Arrays. Finally, we show some other versatile properties of the Shared Paged Binding Arrays and present our conclusions.

2 Speculative Computation in Non-deterministic Dependent Goals

Unrestricted and-parallel execution of dependent goals with non-determinism can lead to large amount of speculative computation. Consider the goal `p(X), q(X)`, in which the non-deterministic goals `p` and `q` have a data-dependency due to shared variable `X`[1]. Suppose they are defined by the following clauses:

$$
\begin{aligned}
&p(X) \text{ :- } g_1,\ X = 1,\ s_1.\\
&p(X) \text{ :- } g_2,\ X = 2,\ s_2\\
&p(X) \text{ :- } g_3,\ X = 3,\ s_3.\\
&q(3) \text{ :- } h_1.\\
&q(4) \text{ :- } h_2.\\
&q(5) \text{ :- } h_3.
\end{aligned}
$$

where g_i and h_i involve a fair amount of computation. Clearly, there is only one solution for `X`, namely, 3. In a sequential execution, only h_1 will ever get executed, and that too only once when `X` in `q(X)` is bound to 3 by `p(X)`. However, in unrestricted and-parallel execution of `p(X)` and `q(X)`, `p` will produce three bindings for `X` (`X = 1`, `X = 2`, and `X = 3`). So will `q` (`X = 3`, `X = 4`, and `X = 5`) and in the process it will also execute the goals h_1, h_2, and h_3. When the bindings produced by `p` and `q` are compared, only `X = 3` will produce an answer, the rest will be thrown away. Thus compared to sequential execution, dependent and-parallel execution performed redundant execution of h_2 and h_3.

However, if we are too cautious, and execute `q` only after a binding for `X` has been produced by `p` then we may lose parallelism, because for a different set of instantiations all bindings for `X` in `q` may be consistent with those in `p`. Therefore, there is the problem of striking a balance between the amount of speculative computing and the amount of parallelism exploited since the speculativeness or usefulness of a computation depends on the nature of instantiation of variables in the goal (note that the problem of determining variable instantiations in advance is undecidable).

In general, we believe that to extract maximum parallelism and obtain the best possible execution, programs will need suitable (but simple!) explicit control constructs (such as the soft and hard sequencing

[1]Following [28] we term `X` the dependent variable.

operators [20], wait guards [21], etc.), as well as implicit control constructs such as suspension on variables [33, 21, 20, 9, 6, 29, 32]. In this paper we do not pursue this problem further since it is not our main concern.

3 Environment Representation in presence of Non-determinism

Given a non-deterministic goal, its or-parallel execution will entail executing the bodies of more than one matching clauses in parallel. If an unbound variable exists in the goal, then each one of the matching clauses may impart a binding to this unbound variable distinct from others. In other words, during or-parallel execution multiple environments may exist at run-time, each corresponding to a branch in the SLD tree (search tree) of the goal. Efficient representation and maintenance of these multiple environments is a difficult problem [36, 18].

3.1 Problems in Environment Representation

In principle, or-parallelism should be easy to implement since various branches of the search tree are independent of each other, thus requiring little communication between processors. However, in practice, implementation of or-parallelism is difficult because of the sharing of nodes in the search tree.

Consider two nodes in two different branches of search tree. All nodes above (and including) the least common ancestor node of these two nodes are (logically) shared between the two branches. If a binding for a variable, created in one of the common ancestor nodes, is generated above (or at) the least common ancestor node then this binding is the same for both branches and hence should be shared (such a binding is known as an unconditional binding and such a variable as an unconditional variable). However, if a binding to a variable created in a shared ancestor node is generated by a node below the least common ancestor node then that binding should be visible *only to the branch to which the binding node belongs* (such a binding is known as a conditional binding and such a variable as a conditional variable). Bindings of such variables should not be shared. It is this peculiar nature of or-parallel execution, that part of the environment is to be (logically) shared between different branches while part of it is to be kept (logically) distinct, that makes efficient representation of multiple environments so difficult.

Consider the goal `p(X)` and the clauses for `p` defined in section 2. The three different clauses (each giving rise to a different branch in the search tree) each impart different bindings to the conditional variable `X`. Unlike in sequential execution a binding for `X` can't be recorded in the activation record (environment frame) created for the goal since there are many such bindings. Therefore, special arrangements have to be made. If `X` is accessed in the goal s_i then the environment representation scheme has to make sure that the appropriate binding is accessed for `X`. Thus, the main problem in implementing or-parallelism is the efficient representation of multiple environments that co-exist simultaneously in the search tree corresponding to a program. Note that the main problem in management of multiple environments is that of efficiently representing and accessing the conditional bindings; the unconditional bindings can be treated as in normal sequential execution of logic programs (i.e., they can be stored in-place in the activation frames).

3.2 Criteria for Environment Representation in Or-parallel Execution

Essentially, the problem of multiple environment management has to be solved by devising a mechanism where each branch has some private area where it stores conditional bindings applicable to itself. There are many ways of doing this [36, 18]. For example:

- storing the conditional binding created by a branch in an array or a hash table private to that branch, from where the binding is accessed whenever it is needed.
- keeping a separate copy of the environment for each branch of the tree, so that every time branching occurs at a node the environment of the old branch is copied to each new branch.
- recording all the conditional bindings in a global datastructure and attaching a unique identifier with each binding which identifies the branch a binding belongs to.

Each approach has its associated cost. This cost is non-constant time and is incurred either at the time of variable access, or at the time of node creation, or at the time a processor begins execution of a new branch. In [18] three criteria were derived for an ideal or-parallel system, namely:

1. The cost of *environment creation* (also termed task creation) should be constant-time;
2. The cost of *variable access and binding* should be constant-time; and
3. The cost of *task switching*[2] should be constant-time.

[2]That is, the cost associated with updating the state of a processor when it switches from one node of the tree to another.

It has been shown that it is impossible to satisfy these three criteria simultaneously [18]. At least one of the three criteria has to be sacrificed. In other words, the non-constant time costs in managing multiple or-parallel environments cannot be avoided. However, one can argue that systems that choose to have non-constant time task-switching are better than those that choose to have non-constant time task-creation or variable access for the following reasons:

a. The number of tasks created and the number of variable-accesses depends on the program, whereas the number of task-switches done during execution depends on the work-scheduler. Clearly, the system implementor has no control over the number of tasks created or variables accessed. However, when, and how often, task-switching is done can be controlled by the system implementor (for example, by controlling task granularity) by building a smart scheduler.

b. Experience gained from the implementation of or-parallel systems such as Muse [2] and Aurora [10] has shown that the quality of the work scheduler has greatest influence on the performance of an or-parallel system. Scheduling involves selecting the most promising node in the search tree from where to pick work (taking into account *speculativeness* of nodes, task granularity considerations, etc.). The operation of picking the most promising node requires searching how much work is available at different nodes, and in general will not be a constant time operation. (this is irrespective of how the information about the work available at each node is kept, i.e., whether in the node itself or in a global datastructure separate from the search-tree). Thus, a scheduler that picks most promising work at any given time will incur non-constant time cost. Thus, an or-parallel system that has non-constant time task-creation or non-constant variable access will incur non-constant time cost during task-switching as well if it desires to have smart scheduling (note that task-switching involves invoking the scheduler). Note that in an or-parallel system with non-constant time task-switching, the cost of scheduling will be amortized in the task-switching overhead (for example, in Aurora, the updating of the binding array, the cause of non-constant time task switching, is done while looking for most promising work [31, 30, 3]).

The observations above are also borne by experimental results of Szeredi [31] for the Aurora system [10], a system with non-constant time task-switching. He found that the actual cost of updating the binding array is not a very significant part of the total cost incurred in task-switching, and that the most significant portion of the task-switching overhead was incurred in looking for node with most promising work. This overhead will be present in *any* or-parallel system that includes a good scheduler.

In the presence of and-parallelism the problem of environment representation is compounded further, because not only we have to worry about which bindings are shared and which are distinct but we also have to worry about synchronizing access to variables when two and-parallel goals attempt to bind the same variable, and making sure that the environment of one and-parallel goal is 'visible' to the other.

Based on the above, we conclude that the environment representation technique used for realizing or-parallel execution (i) Should have constant-time variable access and task-creation, and non-constant time task-switching (Binding Arrays [37, 38] and Stack Copying [2] are two such environment representation approaches[3]) and (ii) should be able to work even in the presence of IAP or DAP. In this paper we present such an environment representation scheme called the Shared Paged Binding Arrays.

3.3 Earlier Solutions to the Environment Representation Problem

Due to the difficult nature of solving the environment representation problem initial efforts for parallelizing logic programming languages either focussed on designing abstract models of parallel execution, ignoring the implementation issues, or did not take full advantage of the sequential implementation technology that has been developed for logic programming (such as the WAM [34]), or banished non-determinism altogether. An example of the last approach are the committed-choice languages [6, 32, 27, 26]. These languages did not banish non-determinism altogether, but retained it partially in the form of *deep guards*. However, when it came to parallel implementation, a further restriction was placed on the deep-guards—they could only be tried sequentially, or they could only be deterministic. Eventually, the deep-guards gave way to *flat* guards. The point we want to make is that lack of efficient environment representation techniques has been a severe hurdle in realizing fully general parallel logic programming systems in the past.

In the past few years, many researchers have proposed schemes for environment representation [8, 39, 2, 37, 38] (See [18] for a comprehensive comparative survey of these schemes) in or-parallel execution of logic programs. However, these scheme are:

[3]The discussion above also explains why Aurora, based on Binding Arrays, and Muse, based on Stack Copying, are the two most successful or-parallel systems. Performance of systems that use techniques with constant-time task switching, such as ROPM [25], ParaAKL [23], and, to an extent, PEPSys [39], is not up to the mark because of precisely the reasons just given.

a. either too inefficient because they do not take advantage of sequential implementation technology: such schemes are based on the *process model* of execution. In the process based approaches, prominent examples of which are Conery's And-Or Process Model [8] and the Reduce-Or Process Model [25], a process is created for every goal encountered during execution (in contrast, in multisequential approaches, multiple WAM like threads are created that are executed in parallel to produce answers to a query). These processes communicate bindings and control information to each other to finally produce a solution to the top-level query. Process based approaches have also been used for implementing committed choice languages [27].

To be competitive, the performance of a parallel system on a single processor should be comparable to sequential logic programming implementations (ideally there should not be any slow down compared to a sequential system, but most parallel systems will have some overheads). To achieve this goal, parallel systems should take full advantage of the well-developed sequential compilation technology [35]. Experience has shown that process based system do not achieve this. Moreover, presence of backtracking makes the process model too complex for non-deterministic parallel logic programming (although the approach is still good for committed choice languages where there is no backtracking involved). Finally, the process based approach exploits parallelism at a level that is too fine grained, resulting in high parallel overhead and unpromising absolute performances.

b. or, they choose to have non-constant time task creation or non-constant time variable access. Methods of this class were proved to be inferior to methods that have non-constant time task switching in the previous section.

c. or, they are targeted towards exploiting *only* or-parallelism and break down when and-parallelism is present.

With recent advances in design of abstract models for parallel execution of logic programs (such as the Andorra Model [33, 21, 20, 9] and others [28]) which exploit all three forms of parallelism, the need for an environment representation scheme that can be used for implementing these models is being felt more than ever. We believe that the *Shared Paged Binding Arrays* that we propose in this paper will fulfill these needs. The Shared Paged Binding Array owes much to the recent advances made in representing environment in purely or-parallel logic programming systems [8, 38, 2, 19, 11, 14, 4]. As we will shortly demonstrate, the Shared Paged Binding Array (SPBA for

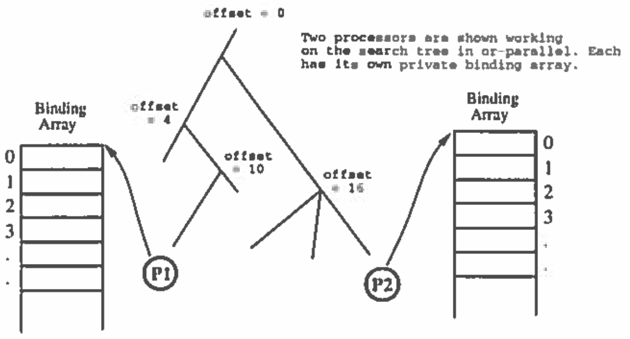

offset = 0
Two processors are shown working on the search tree in or-parallel. Each has its own private binding array.
Binding Array
offset = 4
offset = 10
offset = 16
Binding Array
0
1
2
3
P1
P2
0
1
2
3

Figure 1: Binding Arrays

brevity) can be used for environment representation in any arbitrary parallel logic programming system (that is, any arbitrary combination of or-, independent and-, and dependent and-parallelism), as well as for implementation of Committed Choice Languages with Deep Guards.

4 Simple Binding Arrays

Or-parallel execution can be viewed as parallel unfolding of a *search tree*[4]. Each branch node of the search tree is a *choicepoint*, and represents a set of alternative matching clauses for some goal. Each branch of the search tree represents the set of single-matching clause reductions between two choicepoints. Processing agents can therefore be seen as placed in a node of the search tree, which they unfold by expanding one of the outgoing branches, and possibly by creating new nodes.

As mentioned earlier, the principal problem when implementing or-parallelism is that of managing multiple bindings for variables. The *Binding Arrays* method [38, 37] has proven very successful in managing conditional bindings efficiently in purely or-parallel systems [10], and is outlined below.

Binding arrays are simple one dimensional arrays. One binding array is allocated for each processor of the multiprocessor system. Moreover, a counter is associated with each node in the search tree of the program. The value of the counter is initialised to 0 at the root node. Whenever branching takes place in the (or-parallel) search tree the parent node's counter is copied into the children nodes. Whenever a variable that might receive more than one binding is created by a processor, it is tagged as conditional, and the value of the counter, say i, is recorded in that variable. This recorded value is called the *variable's offset*. The counter for that branch is then atomically incremented.

[4]The search tree is also called the or-parallel tree. An example or-tree is shown in Figure 1.

The ith location in the binding array of the processor is allocated for holding the binding for the variable with offset i. Note that one step of indirection is therefore introduced in dereferencing a variable—when a conditional variable is to be dereferenced, its offset is obtained and used to index into the binding array to access the location where the binding is to be recorded/found (fig. 1). But like in sequential logic programming implementations, access to a variable's value remains a constant-time operation—the principal advantage of the BA method over other methods (see the dereferencing algorithm given below).

```
/*unbound variables are bound to themselves*/
deref(V)
term *V {
    if V→tag == VAR
            if notV→value == V
                deref(V→value)
            else  V
    else    if V→tag == NON-VAR
                V
            else {    /*conditional var bound to offset i*/
                val = BA[i];  /*BA is the binding array.*/
                if val→value == val
                V
                else deref(val) }}
```

However, with this advantage comes the requirement that the BA of a processor should reflect the correct environment of the node at which the processor is currently stationed. Therefore, when a processor switches from one node of the search tree to another, it has to update its BA so that it reflects the environment of its target node. For this purpose, conditional bindings are recorded in a *trail stack* along with the addresses of the corresponding variables. Processors update their BAs from the trail stack as they move up and down the or-parallel tree while searching for nodes with unexplored alternatives.

5 Environment Representation in And-Or Parallel Systems

In this section we present an extension of the simple BAs called the Shared Paged Binding Arrays, but before we do that: (i) we introduce models for abstractly representing and-or parallel execution; (ii) resolve the conflicting requirements imposed by and- and or-parallelism on processor organization; and, (iii) show why simple BAs fail to work in the presence of and-parallelism.

5.1 Abstract Representation of And-Or Parallel Execution

Before we delve into the implementation of and-or parallel logic programming systems, we first need to have an abstract representation for and-or parallel execution (like the or-parallel tree for or-parallel execution).

In and-or parallel systems there are two choices with respect to executing a conjunction of two non-deterministic (dependent or independent) goals p & q[5]: For every solution produced by p compute q in its entirety (also known as and-or parallelism with goal recomputation); or, compute solutions for p and q and then (incrementally) construct their join (also known as and-or parallelism with goal reuse). We have proposed the Extended And-Or Tree [19] for abstractly representing and-or parallel execution with goal reuse and the Composition-tree for representing and-or parallel computation with goal recomputation [15]. Examples of these are shown in figure 2.(i) and 2.(ii). These two abstract representations are very similar to the or-tree representation except that when and-parallel goals are encountered a *parallel conjunct node* (rectangular box in figure 2.(i)-(ii)) is created. The execution tree for each and-parallel goal is then rooted at this parallel conjunct node (Figure 2). The execution tree for the continuation of the and-parallel goals is appended to the parallel-conjunct node in the extended and-or tree (i.e., when goal reuse is adopted), and is rooted below the composition-subtree of one of the and-parallel goals in the case of composition-tree (i.e., when goal recomputation is adopted; in figure 2.(ii) it is put below the subtree of rightmost and-subgoal) [6]. These two representations (that use goal reuse and goal recomputation respectively) are sufficient for abstractly representing any and-or parallel execution where the and-parallelism may be either IAP or DAP.

5.2 And-Or Parallelism and Teams of Processors

In an and-or parallel logic programming system the following problem arises: For efficient execution, two processors that are executing in and-parallel should see all the bindings created by each other so that wasteful computation is not performed by one due to lack of knowledge of values of variables created by the other (as discussed in the example in section 2). This is in contrast to what is needed for or-parallelism where processors working in or-parallel should not see the (conditional) bindings created by each other. Thus, the requirements of or-parallelism and and-parallelism seem anti-thetical to each other. The solution that have been proposed (mostly for independent and- and or-parallel systems) range from

[5]Throughout this paper p & q indicates that the goals p and q may be executed in and-parallel.

[6]Note that one can have goal reuse or goal recomputation within the same computation—the execution tree can be abstractly represented as a combination of the extended and-or tree and the composition tree

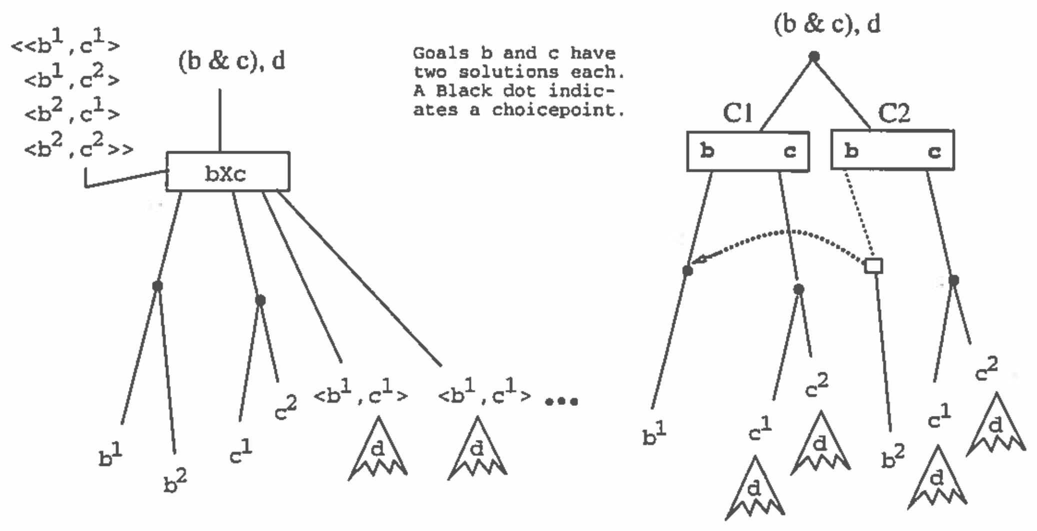


Fig (i): Extended And-Or Tree   Fig (ii): Composition Tree

The Extended And-Or Tree shown above is used for representing and-or parallel computation with goal reuse. That is, when a parallel conjunct is encountered, the rectangular node is created. Solutions are found for goals in the parallel conjunct, which are then cross-produced. The continuation of the parallel conjunct is executed for each element tuple in the cross-product set constructed.

The Composition tree shown above is used for representing and-or parallel computation with goal recomputation: for each solution for the goal to the left in the parallel conjunct, the goals to the right are re-executed. A parallel conjunct (the rectangular box in (ii)) represents a set of solutions. In the tree above a maximum of four such rectangular boxes could have been created, each corresponding to one solution for (b&c). The number of boxes created depends on the processing resources available. In the tree above, solutions of goal c will be found via backtracking assuming that not enough resources are present. The dashed arrow above indicates that C1 and C2 share branches up to the root of the tree

Figure 2: Abstract Representation of And-Or Parallel Execution

updating the environment at the time independent and-parallel computations are combined [25, 19] to having a complex dereferencing scheme [39]. All of these operations have their cost.

We contend that this cost can be eliminated by organising the processors of a multiprocessor system into *teams*, such that and-parallelism is exploited between processors within a team while or-parallelism is exploited between teams. Thus a processor within a team would behave like a processor in a purely and-parallel system while all the processors in a given team would collectively behave like a processor in a purely or-parallel system. This entails that all processors within each team share the data structures that are used to maintain the separate or-parallel environments. For example, if binding arrays are being used to represent multiple or-parallel environments, then only one binding array should exist per team, so that the whole environment is visible to each member processor of the team. Note that in the worst case there will be only one processor per team. Also note that in a team setup a processor is free to migrate to another team as long as it is not the only one left in the team.

The concept of teams of processors has been successfully used in the Andorra-I system [9], which extends an or-parallel system to accommodate (determinate) dependent and-parallelism. It has also been used in independent and- and or-parallel systems [14, 11] We firmly believe that the organization of processors into teams is absolutely essential for design-simplicity and execution-efficiency of and-or parallel systems.

5.3 Simple Binding Arrays and And-Or Parallelism

The 2 main properties of the BA method for or-parallelism are the following:

(i). The offset of a conditional variable is fixed for its entire life.

(ii). The offsets of two consecutive conditional variables in an or-branch are also consecutive.

The implication of these two properties is that conditional variables get allocated space consecutively in the binding array of a given processor, resulting in optimum space usage in the BA. This is important because large number of conditional variables can exist at runtime[7].

In the presence of and-parallel goals, each of which have multiple solutions, maintaining contiguity in the BA can be a problem, especially if processors are allowed (via backtracking or or-parallelism) to search for these multiple solutions.

Consider the goal: `a, (b & c), d`. The part of its tree corresponding to the parallel conjunction is shown in figure 3.(i) (the figure also shows the number of conditional variables that are created in different parts of the tree). If `b` and `c` are executed in independent and-parallel by two different processors P1 and P2, then assuming that both have private binding arrays of their own, all the conditional variables created in branch `b-b1` would be allocated space in BA of P1 and those created in branch of `c-c1` would be allocated space in BA of P2. Before P1 or P2 can continue with `d` after finding solutions `b1` and `c1`, their binding arrays will have to be merged somehow. In AO-WAM [19] the approach taken was that one of P1 or P2 would execute `d` after updating its Binding Array with conditional bindings made in the other branch (known as the BA loading operation). The problem with BA loading operation is that it acts as a sequential bottleneck which can delay the execution of `d`, and reduce speedups. To get rid of the BA loading overhead we can have a common binding array for P1 and P2, so that once P1 and P2 finish execution of `b` and `c`, one of them immediately begins execution of `d` since all conditional bindings needed would already be there in the common BA. This is consistent with our discussion in section 5.2 about having teams of processors where all processors in a team would share a common binding array.

However, if processors in a team share a binding array, then backtracking can cause inefficient usage of space, because it can create large unused holes in the BA. Referring again to the tree shown in figure 3.(i), when processors P1 and P2 start executing goals `b` and `c`, then to be able to efficiently utilise space in their common binding array they should be able to exactly tell the amount of space that should be allocated for conditional variables in branches `b-b1` and `c-c1`. Let us suppose we know the exact number of conditional variables that arise in branches `b-b1` and `c-c1` (say k_1 and k_2 respectively). If n is the value of the offset counter at the parallel conjunct node C1, then P1 would set its offset counter to n and allocate space between locations n and $n+k_1-1$ in the BA for conditional variables in branch `b-b1`. P2 would set it to $n+k_1$ and allocate space between locations $n+k_1$ and $n+k_1+k_2-1$ in the BA for conditional variables in branch `c-c1` (figure 3.(ii)). However, if P1 fails and tries an untried alternative in one of the choice-points created along branch `b-b1`, and if this new alternative leads to creation of more number of conditional variables than k_1, then there is going to be a problem since P1 will have to get a new value of the offset-counter which does not interfere with that of P2 (i.e., it does not lead to overwriting of BA locations between $n+k_1$ and $n+k_1+k_2-1$). Maintaining the offset counter such that it does not interfere with BA locations being used by P2 can be very complex. Also once P1 has completely searched or backtracked over goal `b` it will be difficult to reuse the free space in the BA created by backtracking of P1 since if P2 has not completely backtracked over `c` yet, then the space after location $n+k_1$ would still be in use, trapping the space freed by P1. Even if we manage to reuse the space freed, it will lead to BA offsets of consecutive CVs in a branch of the tree becoming non-monotonic. This has the determental effect that the offsets, traditionally used as indicators of "seniority" of CVs [LWH90], cannot be used any longer for determining their "age".

To simplify the book-keeping of the offset-counter, one can determine the maximum number, m, of conditional variables created among all branches of the goal `b` and then set P1's counter to n and P2's counter to $n+m$. In this case space would be definitely wasted, since all the m allocated locations would be used by only one of the branches of `b`. The others branches of `b` may have far fewer conditional variables than m.

Unfortunately, knowing the exact number of conditional variables that are going to arise in a branch (k_1, k_2, m, etc.) is impossible since their number is only determined at runtime; the best one can do is to estimate them. Underestimating their number can lead to lot of small holes in the BA which will be used only once (figure 4.(i)), while overestimating them can lead to allocating space in the BA which will never be used (figure 4.(ii)). Thus, in either case space is wasted and a very large binding array may be needed to exploit independent and-parallelism with or-parallelism.

5.4 Shared Paged Binding Array

To solve the above problem we divide the binding array into *fixed sized segments*. Each conditional variable is bound to a pair consisting of a segment number and an offset within the segment. An auxiliary array keeps track of the mapping between the segment number and its starting location in the binding array. Dereferencing CVs now involves double indirection: given a conditional variable bound to $\langle i, o\rangle$, the starting address of its segment in the BA is first found from location i of the auxiliary array, and then the value at offset o from that address is accessed. A set of CVs that have been allocated space in the same logical segment (i.e., CVs which have common i) can reside in any physical page in the BA, as long as the

[7] For instance, in Aurora [10] more than 1Mb of space gets allocated for each BA.

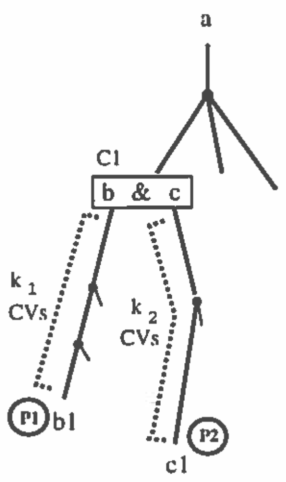


Fig (i): Part of a C-tree

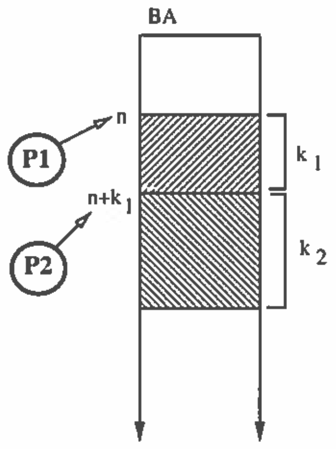


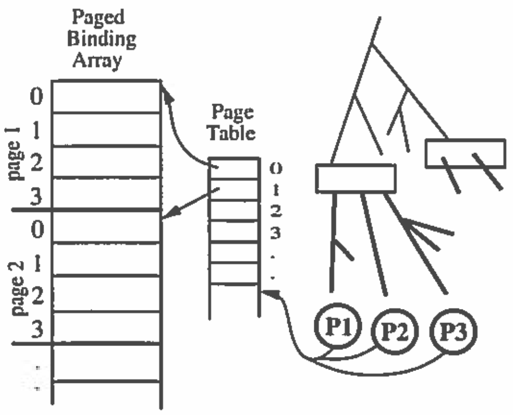
Figure (ii): Optimal Space Allocation in the BA

Figure 3: BAs and And-Parallelism

starting address of that physical page is recorded in the *i*th slot in the auxiliary array. Note the similarity of this scheme to memory management using paging in Operating Systems, hence the name Paged Binding Array (PBA)[8]. Thus a segment is identical to a page and the auxiliary array is essentially the same as a page table. The auxiliary and the binding array are common to all the processors in a team. From now on we will refer to the BA as Paged Binding Array (PBA) and the auxiliary array as the Page Table (PT)[9]. Since a PBA is shared between all member processors of a team, we call it *Shared Paged Binding Array* or SPBA.

Every time execution of an and-parallel goal is started by a processor, or the current page in the SPBA being used by that processor for allocating CVs becomes full, a ***page-marker node*** containing a unique integer id *i* is pushed onto the trail-stack (Figure 5). The unique integer id is obtained from a shared counter (called `pt_counter`); there is one such counter per team. A new page is requested from the SPBA, and the starting address of the new page is recorded in the *i*th location of the Page Table. *i* is referred to as the page number of the new page. Each processor in a team maintains an offset-counter, which is used to assign offsets to CVs within a page. When a new page is obtained by a processor, its offset-counter is reset. Conditional variables are bound to the pair <*i*, *o*>, where *i* is the page number, and *o* is the value of the offset-counter, which indicates the offset at which the value of the CV would be recorded in the page. Every time a conditional variable is bound to such a pair, the offset counter *o* is incremented. If the value of *o* becomes greater than K, the fixed page size, a new page is requested and a new page-marker node is pushed.

A list of free pages in the SPBA is maintained separately (as a linked list). When a new page is requested, the page at the head of the list is returned. When a page is freed by a processor, it is inserted in the free-

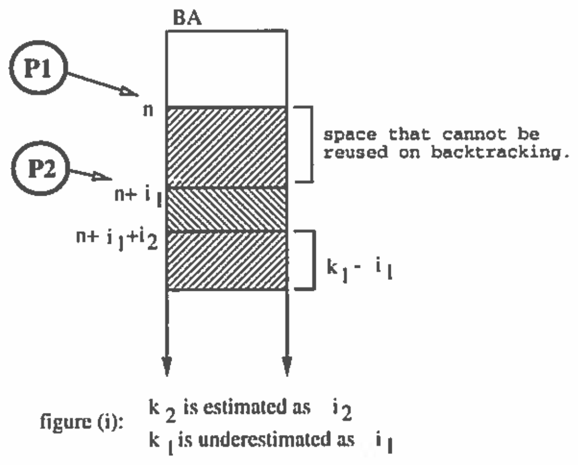


An and-parallel computation (delimited by a rectangular box) is performed by a team of three processors which share a common paged binding array and page table. The branches that are part of and-parallel computation are shown in dark in the figure.

Figure 5: Paged Binding Arrays

[8]Thanks to David H. D. Warren for pointing out this similarity.

[9]A binding array has also been used in the ElipSys system of ECRC [1], but for entirely different reasons. In ElipSys, when a choice point is reached the BA is replicated for each new branch. To reduce the overhead of replication, the BA is paged. Pages of the BA are copied in the children branches on demand, by using "copy-on-write" strategy. In ElipSys, unlike our model, paging is not necessitated by independent and-parallelism.


BA
P1
n
P2
n+ i1
n+ i1+i2
space that cannot be reused on backtracking.
k1 - i1
figure (i): k2 is estimated as i2
k1 is underestimated as i1

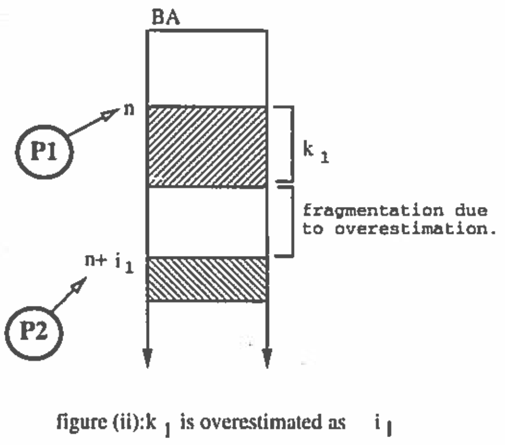

BA
P1
n
k1
fragmentation due to overestimation.
n+ i1
P2
figure (ii):k1 is overestimated as i1

Figure 4: Problems with Simple BA

list. The free-list is kept ordered so that pages higher up in the SPBA occur before those that are lower down. This way it is always guaranteed that space at the top of the SPBA would be used first, resulting in optimum space usage of space in the SPBA. The algorithms for dereferencing and creation of a conditional variable are given below:

```
/*K is max. no. of slots in a page*/
create(V)
term *V {
  if (oc > K) ∨ (new and-parallel goal is begun) {
    oc = 0;
    p = head(free_page_list);
    i = pt_counter;
       /*exclusive access to pt_counter is obtained.*/
    pt_counter++;
    PT[i] = p;      /*PT is the Page Table.*/
    push_page_marker(i) }
  V = <i, oc>;
  oc = oc+1}


/*unbound variables are bound to themselves*/
/*conditional vars bound to <i, o>*/
/*SPBA is the paged binding array.*/
deref(V)
term *V {
    if V→tag == VAR
            if not V→value == V
                  deref(V→value)
            else V
    else    if V→tag == NON-VAR
                  V
            else {
                  val = SPBA[o + PT[i]];
                     /*conditional vars bound to <i, o>*/
                  if val→value == val
                        V
                  else deref(val) }}
```

Note that the SPBA would allow backtracking without any space in it being used only once or never being used (the problems mentioned in section 5.3). However, some internal fragmentation can occur in an individual page, because when a set of conjunctive goals to be executed in and-parallel is encountered, conditional variables in each of its and-parallel goal are allocated space in a new page, and so if part of the current page is unused, it will be wasted (it may, however, be used when backtracking takes place and another alternative is tried). Given that the granularity of goals in and-parallelism is preferred to be large, we expect such wastage as a percentage of total space used to be marginal.

Returning to the tree shown in Figure 3.(i), when the parallel conjunct node is reached, processor P1 will request a new page in the SPBA and push a marker node in the trail-stack. The conditional variables created in branch `b-b1` would be allocated space in this new page. In the meantime P2 would request another page from the SPBA where it will allocate CVs created in branch `c-c1`. P1 and P2 may request more pages, if the number of CVs created in respective branches is greater than K. If the branch `b-b1` fails, then as P1 backtracks, it frees the pages in the SPBA that it allocated during execution. These pages are entered in the free-list of pages, and can now be used by other processors in the team, or by P1 itself when it tries another alternative of `b`.

Note that with the SPBA, when a team picks an untried alternative from a choice point, it has to task-switch from the node where it is currently stationed to that choice point. In the process it has to install/deinstall the conditional bindings created along that path, so that the correct environment at that choice point is reflected (like in Aurora, a conditional binding made to a variable is trailed in the trail-stack, along with address of that variable). While installing conditional bindings during task-switching, if a team encounters a page-marker node in the trail-stack whose id is j, it requests a new page in the SPBA from the free list, records the starting location of this page in jth location of the page table, and continues. Likewise, if it encounters a page-marker node in the trail-stack during backtracking whose id is k, it frees the page being pointed to by the kth location of the page table.

If the untried alternative that is selected is not in the scope of any and-parallel goal, then task-switching is more or less like in purely or-parallel system (such as Aurora), modulo allocation/deallocation of pages in the SPBA. If, however, the untried alternative that is selected is in the scope of an and-parallel goal **g**, then the team updates its SPBA with all the conditional bindings created in the branches corresponding to goals which are to the left of **g**.

Note that the technique of paged binding array is a generalisation of environment representation technique of AO-WAM [19], hence some of the optimisations [17] developed for the AO-WAM, to reduce the number of conditional bindings to installed/deinstalled during task-switching, will also apply to the SPBA model. Also note that the SPBA is a generalization of the environment representation scheme used in the Andorra-I system [9] that exploits or-parallelism and deterministic dependent and-parallelism. Lastly, seniority of conditional variables, which needs to be known so that "older" variables never point to "younger ones", can be easily determined with the help of the $<i,\ o>$ pair. Older variables will have a smaller value of i; and if i is the same, then a smaller value of o.

6 And-Or Parallel Systems and SPBAs

We now describe how different systems that combine or-parallelism with different flavours of (dependent and independent) and-parallelism can be realized using the SPBA. For each kind of system only a rough outline is sketched—details are omitted due to lack of space. The SPBAs can also be used for implementing Parallel CLP Systems [12].

6.1 Combined IAP and OP Systems

Implementation of systems that combine Or-parallelism and Independent and-parallelism using the shared paged binding arrays is quite simple. It is customary to express independent and-parallelism through *Conditional Graph Expressions* [22], which determine if a conjunction of goals can be executed in independent and-parallel or not through simple run-time checks . Note that, as mentioned, and-parallelism is exploited by processors within teams, while or-parallelism is exploited between teams. Since the goals are independent, no special action needs to be taken after the execution of independent and-parallel goals is over.

6.1.1 Combined IAP and OP Systems with Goal Recomputation

In this case, a composition tree is constructed. There are three distinct cases that should be considered: (i) when a parallel conjunct is encountered by a team; (ii) when an alternative from within a goal in a CGE is picked by another team; and, (iii) an untried alternative in continuation of the parallel conjunct is picked. Regarding case (i), when the parallel conjunct is encountered a parallel conjunct node is created. For example, in the tree shown in fig. 2.(ii) a parallel conjunct node is created by the processor P1. P1 will request a new page in the PBA and push a marker node in the trail-stack. The CVs created in branch `b-b1` by P1 would be allocated space in this new page. In the meantime processor P2, teammate of P1, will pick up goal `c` for execution and would request another page from the PBA where it will allocate CVs created in branch `c-c1`. P1 and P2 may request more pages, if the number of CVs created in respective branches is greater than the size of the page. If the branch `b-b1` fails, then as P1 backtracks, it frees the pages in the PBA that it allocated during execution. These pages are entered in the free-list of pages, and can now be used by other processors in the team, or by P1 itself when it tries another alternative of `b`. The continuation of the independent and-parallel conjunct will be begun by a team when a solution has been found for all goals in the parallel conjunct. Note that as soon as a solution is found for all goals in the parallel conjunct, execution of its continuation can begin since all the bindings needed are already present in the SPBA.

In case (ii), when a team picks an untried alterative from a choice point in an and-parallel goal, the team first task switches to that node, updating its SPBA and the Page Table along the way. The solution to the alternative picked is found, and then the subgoals to the right of this subgoal in the parallel conjunct are re-executed (goal recomputation). Execution from then on is similar to what is described in the previous paragraph.

In case (iii), again the team will have to task switch to the node in the continuation that has available work. In the course of task-switch, the parallel conjunct will be encountered and the SPBA and Page Table updated.

The implementation of combined or- and independent and-parallel systems (with or without recomputation of goals) is quite simplified with Shared Paged Binding Arrays. An implementation of an or- and independent and-parallel system with recomputation of goals based on SPBA has the nice property that in the presence of only or-parallelism the system will behave exactly like a purely or-parallel system based on simple binding arrays, such as Aurora [10] (modulo an extra level of indirection in accessing conditional bindings). In the presence of only and-parallelism it will behave exactly as a purely independent and-parallel system such as &-Prolog [22] (modulo the indirection in accessing conditional variables).

6.1.2 Combined IAP and OP Systems with Goal Reuse

In this case an extended and-or tree will be constructed. Here again, three cases can be distinguished. (i) when a parallel conjunct is encountered by a team; (ii) when an alternative from within a goal in a CGE is picked by another team; and, (iii) an untried alternative in continuation of the parallel conjunct is picked. Regarding Case (i), a parallel conjunct node will be created by a team when an independent and-parallel conjunct is encountered. Processors in the team pick up goals in the parallel conjunct for independent and-parallel execution. Before commencing the execution of the and-parallel goals, they each request a page for allocating space for conditional variables. When the execution of goals is finished, the team will continue with the execution of the continuation. No extra work is required since all the requisite pages are already in the SPBA. For example, consider figure 2.(i). A team begins execution by creating a parallel conjunct. Processors P1, P2, that are members of the team work on goals b and c respectively. When solutions b_1 and c_1 are found, the cross-product tuple $\langle b_1, c_1 \rangle$ is constructed and execution of the continuation of the conjunct for this tuple is begun.

Regarding Case (ii), the team will first have to task switch to the node containing the alternative, updating the SPBA and the Page Table in the process with conditional bindings created earlier. The team will begin execution of the alternative; other member processors will execute other alternatives in other and-parallel goals. For instance, in the example in Fig. 2.(i) another team may generate b_2 and c_2 while the team that created the conjunct is still generating b_1 and c_1. Once b_2 and c_2 are generated, the other team can generate all cross-product tuples in which b_2 and c_2 take part, and then it can continue execution of the continuation of the and-parallel conjunct for one of these generated tuples, exactly as the first team.

Case (iii) arises when a team simply picks up new tuples, whose components have been generated by other teams. In this case the team has to first taskswitch to the parallel conjunct node, followed by installing all conditional bindings created while finding the solutions that constitute the tuple.

Note that often the conditional bindings of different goals in the parallel conjunct can be themselves installed in parallel. Implementation of the conjunct node itself should also be much simplified, as the node now simply has only to maintain a list of tasks which have unexplored alternatives, so that a team can move there and exploit or-parallelism. The list of cross-product tuples generated is kept in the parallel conjunct node so that that a team can pick untried tuples. We believe that the SPBA should make it practical to implement and-or parallel systems that recompute goals, that share goals, or that can follow both strategies, depending on which is best for a particular procedure.

6.2 Combined DAP and OP Systems

We now describe how the Shared Paged Binding Array can be used for implementing systems with or-parallelism and dependent and-parallelism. Let us consider the goal `p(X) & q(X)`, where p and q are non-deterministic (i.e., have more than one matching clauses) and suppose p and q are executed in parallel. Most systems exploiting dependent and-parallelism will distinguish between goals p and q, designating p as the *producer* of the bindings of X, and q its *consumer*. Only producers can bind an unbound dependent variable. Consumer goals will suspend if they attempt to do so.

Like in independent and-or parallel systems, or-parallelism will be exploited by teams while and-parallelism will be exploited by processors within teams. Execution will begin by a team T1 reaching the goal in question. A processor in the team will begin executing p while another processor will pick up q. Within p and q they will pick one of the alternatives (since p and q are non-deterministic). The variable X would have been allocated space in the shared paged binding array of the team. Since the Paged Binding Array is shared, a processor accessing X (for binding it, or reading its value), will have to obtain exclusive access to it.

As a result of sharing the Binding Array, the moment a binding for X is produced by one of the processors and recorded in the SPBA it is immediately visible to the others in the same team. Thus, other team-members can tell the binding status of each dependent variable with ease.

Other teams may come along and execute other alternatives of p and q while T1 is busy producing the first solution. Here we consider a number of cases:

6.2.1 Combined Determinate DAP and OP Systems

An example of such a system is the Andorra-I system based on the Basic Andorra Model. Note that the Andorra-I system uses Binding Arrays and the notion of teams of processors in its implementation. In Andorra-I, processors in a team share a binding array, each processor in the team allocates a chunk of locations in the common Binding Array in which it allocates conditional variables. Note that since the SPBA is a generalization of the chunk scheme of Andorra-I, the SPBA can be trivially used instead for implementing Andorra-I where size of a chunk equals size of a page. (Note that offsets assigned to CVs in Andorra-I are single values, while if we use SPBA instead, CVs will have to be assigned pairs. The implementation of Andorra-I doesn't need pairs because dependent and-goals are always deterministic, while the SPBA is designed for non-deterministic goals as well.)

6.2.2 Combined DAP and OP Systems with Goal Recomputation

We consider here execution models such as Prometheus [28, 29] which suspend a consumer goal if it attempts to bind the dependent variable, and the consumer goal is executed for every solution of the producer goal. Execution of models such as Prometheus can be abstractly represented quite easily using the composition tree.

Execution takes place by a team first reaching the goal `p(X) & q(X)`. Two processors in the team pick the first alternatives from p and q respectively. The alternative for `q` will be executed so far as the processor executing `q` doesn't attempt to bind `X`. If it attempts to bind `X` it will suspend and will resume only after the alternative for `p` has produced a binding for `X`. Since the team has a common SPBA, checking whether `X` is unbound or bound is quite easy.

While a team is computing the first solution for `p(X) & q(X)`, other teams may pick alternatives from `p`. In Prometheus they will recompute q in its entirety, rather than make use of the computation done by the first team.

A team may also pick alternatives from `q`, but before doing so the team must update its SPBA from the trail in the corresponding solution for `p`. However, all dependent and-parallel goals to the right of `q` (if any) will be completely recomputed.

Prometheus [28] and similar dependent and- and or-parallel systems can be viewed as examples of dependent and-parallel system that extend independent and-parallel systems with recomputation. The SPBA allows easy implementation of the main extension, namely, the mechanism where consumer goals that attempt to bind an unbound dependent variable suspend.

6.2.3 Combined DAP and OP Systems with Goal Reuse

There are other execution models that exploit dependent and-parallelism and or-parallelism and that do not recompute goals (e.g., the Extended Andorra Model (EAM) [33, 20] and the Andorra Kernel Language [21]). In the EAM given the goal `p(X) & q(X)`, `q` is not recomputed for every solution for `p`, rather both `p(X)` and `q(X)` are executed independently. However, an alternative of `q` (the consumer goal) will suspend if it attempts to bind `X` before `p`. The execution of suspended consumer goals is restarted by making a copy of computation state of goal `q` for each binding produced by the producer `p`. This operation, known as the non-determinate promotion, is akin to taking a join of the solutions of the producer and the consumer goals (where the solutions of the consumer have not been found completely; the remaining computation in the consumer goal can be regarded as part of the continuation of the parallel conjunct so that the model becomes somewhat similar to the combined IAP and OP model with goal reuse). However, the consumer `q` is allowed to bind `X` unconditionally if the binding is *deterministic*.

Execution of such models can be abstractly represented using the extended and-or tree. Given such an execution model, execution using SPBA will proceed as follows: A team will pick up the and-parallel goals for execution. A processor in the team will pick the first alternative of `p` and another processor will pick the first alternative of `q`. Conditional bindings made by the two processors in the team will be recorded in the SPBA. If `q` produces a binding for `X` and `X` is unbound, it will suspend unless it is determinate (code for checking if a goal is determinate can be generated at compile-time, say as in [9]). If the processor executing `q` produces a binding that is inconsistent with the binding produced by its teammate processor executing `p`, then the one executing `q` will backtrack. In the process of backtracking it will only untrail bindings from the paged binding array that were produced as a result of execution of `q`. The stack frames, trail, etc., produced by `q` will remain intact (since they may be used by another team that has produced a consistent binding for `X` from `p`). The backtracking processor then either installs bindings from an alternative of `q` that has been executed by another processor and that has a consistent binding for its `X`, or it will try to execute another alternative for `q`. Other teams will behave in a manner similar to the first, their different processors executing `p` and `q` to obtain consistent bindings. Note that deterministic bindings will not be put in the Binding Array, rather they can be put in-place in the environment frames.

We have discussed so far the main ideas of these models. Implementation will be more complex when we consider mechanisms such as the *eager execution of producer goals* in the EAM, or local execution of guards in AKL. In both cases goals that work in and-parallel are not allowed to access everyone else's bindings, thus synchronisation between elements of a team is more complex. In this case, the SPBA serves as a very useful temporary datastructure where processors can store synchonisation information.

6.3 Combined IAP, OP and DAP Systems

As may be obvious by now, there are numerous similarities between systems implementing independent and- and or-parallelism and those implementing dependent and-parallelism and or-parallelism. Also, note that independent and-parallelism is a special case of dependent and-parallelism, therefore the implementation of a system using SPBA that exploits independent and-, dependent and- and or-parallelism does not need any extra machinery beyond what has already been described in the previous sections. We omit the description due to lack of space.

6.4 Committed Choice Languages with Deep Guards

Implementation of Committed choice languages with deep guards (such as Concurrent Prolog) in which or-parallel execution of such guards is permitted can also be implemented quite easily with SPBAs. Execution of a CCL with deep guards requires a model that is very similar to the ones described in sections 6.2.2 and 6.2.3. The only difference is when a commit operator is reached, in which case all choicepoints up to the corresponding parallel conjunct must be removed. Removal of choicepoints is accompanied by cleaning up of parts of the SPBA that correspond to the pruned alternatives.

Execution will commence as follows: a team picks up the top level goal for execution. A parallel conjunct is created and member processors attempt to solve the guards in the respective goals. A new page is used by each processor in the SPBA for executing the goal it selects. An attempt by a processor to bind a variable in the goal while executing the guard will result in suspension. The processor will record this suspension and will attempt to do some other useful work. When the variable on which execution is suspended gets bound, a member processor will resume the execution of the suspended goal.

Meanwhile another team can come along and pick an alternative from one of the deep guards. One of the member processor can start executing the untried alternative after updating the SPBA, while the other processors can pick untried alternatives from deep guards of sibling conjunctive goals. A failure in any of the processors in the team will simply cause that processor to backtrack and try another untried alternative. A success, however, will prompt that processor to prune all other alternatives in that deep guard. Once the alternatives are pruned, the body of the clause will be executed output bindings will be performed.

7 Versatility of the Shared Paged Binding Arrays

The Shared Paged Binding Array is intended for parallel logic programming systems targeted towards Shared Memory Multiprocessors. So far we had one SPBA for each team. We can go one step further and have one SPBA that is shared by *all processors of all the teams.* However, each team has its own individual page table, which indicates the pages that the team is currently using. The pages are allocated in the single Paged Binding Array. There are many advantages to having a single system-wide Paged Binding Array: (i) Space can be utilized much better. (ii) Processors in a team can park their page table at some place in the execution tree, break from the team, go and help other teams, and later reform the team and pick up execution from the point where the page table was parked (the memory space that will be blocked due to parking will be exactly equal to the number of pages in use by that page table); parking of binding arrays is very expensive when there are multiple binding arrays. (iii) The use of Binding Arrays in the style of ElipSys [1] is also possible: duplication of a binding array just involves duplicating the page table; pages get physically copied during a write (copy-on-write), thus ensuring that read-only pages are shared.

We can also distinguish between local conditional variables (those that are allocated space in the local stack in the WAM) and global conditional variables (those conditional variables that are allocated space in the heap in the WAM). To do this we will need a separate trail for global conditional variables and local conditional variables. We will also need a separate page tables for the two kind of conditional variables in each team. However, we can use the same SPBA. The advantage of making this distinction is that during task-switching, in certain cases we only need install bindings from the global trail [17]. As a result, task-switching overhead, the principal source of overhead in the SPBA scheme, is reduced.

The page size in the SPBA can be made equal to the page size of the cache of the multiprocessor system. We expect the page size of pages in the SPBA to be of the order of 64 or 128 words. This is also the range of page sizes of caches. As a result of this matching of sizes of pages, we hope that SPBAs will deliver better cache performance (in terms of successful cache hits).

The page size of the SPBA can also be used for controlling granularity of computation. The idea is that we can estimate the granularity for an and-task through the number of pages the tasks needs. If we are creating many tasks which only fill at most one or two pages before creating new and-tasks, we can assume thet these tasks are fine grained, and easily prevent the system from creating extra tasks. As an example, consider the Fibonacci program:

```
fib(0, 1).
fib(1, 1).
fib(X, Y) :- X > 1,
      ( X1 is X-1, fib(X1, Y1) &
        X2 is X-2, fib(X2, Y2) ),
      Y is Y1+Y2.
```

When executing this program in an SPBA based system, we would create new pages for each recursive call to `fib`, each page containing the variables `X1`, `Y1`, `X2`, and `Y2`. To implement granularity control, the system will notice it is creating many mostly empty pages, and thus stop generating new and-tasks for `fib` until it sees pages are being better used. Notice that we have two ways to control granularity. First, through the total number of pages we can check whether there are sufficient and-tasks and whether we need to create more. Second, through the number of pages per and-task we can notice whether we are only creating fine-grained tasks. The same information can be used to support a top-level scheduler, which dynamically reconfigures teams structures depending on the availability of parallelism.

A Criticism one could advance against SPBA based techniques is that they will not work well on Non-Uniform Memory Access architectures such as the BBN Butterfly. However, this can easily be rectified using *generalized stack copying* [13]. The idea is that whenever locality of reference due to our "share everything" approach causes performance degradation on a subset of processors, we can improve locality by forking over a choicepoint and making a copy of the entire computation for that set of processors in the style of MUSE. Thus the copied part and the original copy will execute in or-parallel with each other. Note that during copying all the structures, including SPBAs, will be copied (the only thing shared is the choicepoint on which forking was done). Since we also exploit and-parallelism, the copying done is in the style of ACE [16] rather than MUSE and is not explained here due to lack of space. The generalized copying technique is based on the observation that "stack copying" based and "binding array" based environment representation techniques, each of which have their own advantages, are two extremes of a spectrum of choices (share everything *vs* share very little) one can make for representing environment, and ideally one should combine the two to get benefits of both.

8 Conclusions

In this paper we presented a versatile data-structure called the Shared Paged Binding Array (SPBA) for representing multiple environments in parallel logic programming systems. The SPBA can be used for implementing any arbitrary parallel logic programming system, that is, for implementing a system that exploits any arbitrary combination of independent and-parallelism, or-parallelism, and dependent and-parallelism.

The technique is quite versatile in that if a system, that exploits all forms of parallelism and that has been implemented using SPBA, is used for executing a program that has only or-parallelism, then its performance will be comparable to a system that exploits only or-parallelism (such as Aurora [10]). Likewise, for executing programs that have only independent and-parallelism or only dependent and-parallelism. In the presence of only one processor, the performance of this parallel system will be comparable to a sequential logic programming system because the SPBA can take full advantage of the sequential WAM technology.

9 Acknowledgements

Thanks are due to Rong Yang, David Warren, Kish Shen of the University of Bristol and Manuel Hermenegildo of University of Madrid for many stimulating discussions. This research was initially supported by U.K. Science and Engineering Research Council grant GR/F 27420. Continued support is provided by NSF Grant CCR 92-11732 and HRD 93-53271, Grant AE-1680, AI-1929 from Sandia National Labs, by an Oak Ridge Associated Universities Faculty Development Award, and by NATO Grant CRG 921318. Vítor Santos Costa would like to thank the Gulbenkian Foundation and the European Commission for supporting his stay at Bristol.

References

[1] A. Véron, J. Xu, et al. Virtual memory support for parallel logic programming systems. In *Proceedings of PARLE'91*, Lecture Notes in Computer Science, pages 365–396. Springer Verlag, June 1991.

[2] K.A.M. Ali and R. Karlsson. The muse or-parallel prolog model and its performance. In *1990 N. American Conf. on Logic Prog.* MIT Press, 1990.

[3] T. Beaumont, S. Muthu Raman, and et al. Flexible Scheduling or Or-parallelism in Aurora: The Bristol Scheduler. In *Parallel Architectures and Languages Europe '92, LNCS 506*, pages 403–420. Springer Verlag, 1991.

[4] R. Butler, E. L. Lusk, R. Olson, and R. A. Overbeek. Anlwam: A Parallel Implementation of the Warren Abstract Machine. Internal report, Argonne National Laboratory, Argonne, Il 60439, 1986.

[5] A. Ciepielewski and S. Haridi. Control of Activities in the Or-Parallel Token Machine. In *International Symposium on Logic Programming*, pages 49–58, Silver Spring, Md, February 1984. Atlantic City, Ieee Computer Society.

[6] K. Clark and S. Gregory. Parlog: Parallel Programming in Logic. *Journal of the Acm*, 8:1–49, January 1986.

[7] J. S. Conery. *The And/Or Process Model for Parallel Interpretation of Logic Programs.* PhD thesis, The University of California At Irvine, 1983. Technical Report 204.

[8] J. S. Conery. Binding Environments for Parallel Logic Programs in Nonshared Memory Multiprocessors. In *International Symposium on Logic Programming*, pages 457–467. San Francisco, Ieee Computer Society, August 1987.

[9] V. Santos Costa, D.H.D. Warren, and R. Yang. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-parallelism. In *Proc. 3rd ACM SIGPLAN PPoPP*, 1990.

[10] E. Lusk et al. The aurora or-parallel prolog system. *New Generation Computing*, 7(2,3), '90.

[11] G. Gupta and V. Santos Costa. And-Or Parallelism in Full Prolog with Paged Binding Arrays. In *LNCS 605, PARLE'92*, pages 617–632. Springer-Verlag, June 1992.

[12] G. Gupta and V. Santos Costa. Parallel Implementation of CLP Systems. Internal report, NMSU, 1992.

[13] G. Gupta, V. Santos Costa, and M. Hermenegildo. Gerneralized Stack Copying. In *Proc. Workshop on Parallel Execution JICSLP'92*, Nov 1992.

[14] G. Gupta and M. Hermenegildo. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *LNCS 569*. Springer-Verlag, 1991.

[15] G. Gupta, M. Hermenegildo, and V. Santos Costa. And-or parallel prolog: A recomputation based approach. *New Generation Computing*, 11(3,4):297–322, 1993.

[16] G. Gupta, M. Hermenegildo, E. Pontelli, and V. Santos Costa. ACE: And/Or-parallel Copying-based Execution of Logic Programs. Tech. report, NMSU, Nov 1993.

[17] G. Gupta and B. Jayaraman. Optimizing And-Or Parallel Implementations. In *1990 N. American Conf. on Logic Prog.*, pages 737–736. MIT Press, 1990.

[18] G. Gupta and B. Jayaraman. Analysis of or-parallel execution models. *ACM TOPLAS*, 15(4):659–680, 1993.

[19] G. Gupta and B. Jayaraman. And-or parallelism on shared memory multiprocessors. *Journal of Logic Programming*, 17(1):59–89, 1993.

[20] G. Gupta and D.H.D. Warren. An Interpreter for the Extended Andorra Model. Internal report, University of Brsitol, 1991.

[21] Seif Haridi and Sverker Janson. Kernel Andorra Prolog and its Computation Model. In *Proc. 7th Int'l Conf. on Logic Prog.* MIT Press, 1990.

[22] M. Hermenegildo and K. Greene. &-Prolog and its Performance: Exploiting Independent And-Parallelism. In *1990 Int'l Conf. on Logic Prog.*, pages 253–268. MIT Press, June 1990.

[23] R. Moolenaar and B. Demoen. A Parallel implementation for AKL. In *PLILP'93*, pages 246–261. LNCS 714, 1993.

[24] G. H. Pollard. *Parallel Execution of Horn Clause Programs.* PhD thesis, Imperial College, London, 1981. Dept. of Computing.

[25] B. Ramkumar and L. V. Kale. Compiled Execution of the Reduce-OR Process Model on Multiprocessors. In *Proc. NACLP'89*, pages 313–331. MIT Press, 1989.

[26] E. Y. Shapiro. A Subset of Concurrent Prolog and Its Interpreter. Technical Report TR-003, ICOT, 1-4-28 Mita, Minato-ku Tokyo 108, Japan, January 1983.

[27] E.Y. Shapiro, editor. *Concurrent Prolog: Collected Papers.* MIT Press, Cambridge MA, 1987.

[28] K. Shen. Exploiting Dependent And-parallelism in Prolog: The Dynamic Dependent And-parallel Scheme. In *Proc. Joint Int'l Conf. and Symp. on Logic Prog.* MIT Press, 1992.

[29] K. Shen. *Studies in And/Or Parallelism in Prolog.* PhD thesis, U. of Cambridge, 1992.

[30] R. Sindaha. The Dharma Scheduler — Definitive Scheduling in Aurora on Multiprocessor Architecture. In *Proc. of ILPS'93*, 1993.

[31] P. Szeredi. Performance Analysis of the Aurora Or-Parallel Prolog System. In *1989 North American Conference on Logic Programming.* MIT Press, October 1989.

[32] K. Ueda. *Guarded Horn Clauses.* PhD thesis, University of Tokyo, March 1986.

[33] D. H. D. Warren. The Extended Andorra Model with Implicit Control. ICLP'90 Parallel Logic Programming workshop.

[34] 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.

[35] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, SRI International, 1983.

[36] D. H. D. Warren. Or-Parallel Execution Models of Prolog. In *Proceedings of TAPSOFT'87.* Springer Verlag, LNCS 250, 1987.

[37] D. H. D. Warren. The SRI Model for OR-Parallel Execution of Prolog—Abstract Design and Implementation. In *Proc. IEEE Int'l Symp. on Logic Prog.*, pages 92–102, 1987.

[38] D. S. Warren. Efficient Prolog Memory Management for Flexible Control Strategies. In *IEEE Int'l Symp. on Logic Prog.*, pages 198–203, 1984.

[39] H. Westphal and P. Robert. The PEPSys Model: Combining Backtracking, AND- and OR- Parallelism. In *IEEE Int'l Symp. on Logic Prog.*, pages 436–448, 1987.

Towards Efficient Parallel Implementation of Concurrent Constraint Logic Programming – The CIAO Model and Other Related Ideas

M. Hermenegildo

Facultad de Informática
Universidad Politécnica de Madrid (UPM)
28660-Boadilla del Monte, Madrid, Spain
herme@fi.upm.es

1 Introduction

We present an informal discussion on some methodological aspects regarding the efficient parallel implementation of (concurrent) (constraint) logic programming systems, as well as an overview of some of the current work performed by our group in the context of such systems. These efforts represent our first steps towards the development of what we call the CIAO (Concurrent, Independence-based And/Or parallel) system – a platform which we expect will provide efficient implementations of a series of *non-deterministic, concurrent, constraint logic programming languages*, on sequential and multiprocessor machines.

CIAO can be in some ways seen as an evolution of the &-Prolog [17] system concepts: it builds on &-Prolog ideas such as parallelization and optimization heavily based on compile-time global analysis and efficient abstract machine design. On the other hand, CIAO is aimed at adding several important extensions, such as or-parallelism, constraints, more direct support for explicit concurrency in the source language, as well as other ideas inspired by proposals such as Muse [1] and Aurora [27], GHC [39], PNU-Prolog [30], IDIOM [16], DDAS [32], Andorra-I [31], AKL [20], and the extended Andorra model [40]. One of the objectives of CIAO is to offer at the same time all the user-level models provided by these systems.

More than a precisely defined design, at this point the CIAO system should be seen as a target which serves to motivate and direct our current research efforts. This impreciseness is purposely based on our belief that, in order to develop an efficient system with the characteristics that we desire, a number of technologies have to mature and others still have to be developed from scratch. Thus, our main focus at the moment is in the development of some of these technologies, which include, among others, improved memory management and scheduling techniques, development of parallelization technology for non-strict forms of independence, efficient combination of and- and or-parallelism, support of several programming paradigms via program transformation, and the extension of current parallelization theory and global analysis tools to deal with constraint-based languages.

We will start our discussion by dealing with some methodological issues. We will then introduce some of our recent work in the direction mentioned above. Given the space limitations the description will be aimed at providing an overall view of our recent progress and a set of pointers to some relevant recent publications and technical reports which describe our results more fully. We hope that in light of the objective of providing pointers, the reader will be kind enough to excuse the summarized descriptions and the predominance in the references of (at least recent) work of our group.

2 Separation of issues / Fundamental Principles

We begin our discussion with some very general observations regarding computation rules, concurrency, parallelism, and independence. We believe these observations to be instrumental in understanding our approach and its relationship to others. A motivation for the discussions that follow is the fact that many current proposals for parallel or concurrent logic programming languages and models are actually "bundled packages", in the sense that they offer a combined solution affecting a number of issues such as choice of computation rule, concurrency, exploitation of parallelism, etc. This is understandable since certainly a practical model has to offer solutions for all the problems involved. However, the bundled nature of (the description of) many models often makes it difficult to compare them with each other. It is our view that, in order to be able to perform such comparisons, a "separation analysis" of models isolating their fundamental principles in (at least) the coordinates proposed above must be performed. In fact, we also believe that such un-bundling brings the additional benefit of allowing the identification and study of the fundamental principles involved in a system independent manner and the transference of the valuable features of a system to another. In the following we present some ideas on how we believe the separation analysis mentioned above might be approached.

2.1 Separating Control Rules and Parallelism

We start by discussing the separation of parallelism and computation rules in logic programming systems. Of the concepts mentioned above, probably the best understood from the formal point of view is that of computation rules. Assuming for example an SLD resolution-based system the "computation rules" amount to a "selection rule" and a "search rule." The objective of computation rules in general is to minimize work, i.e. to reduce the total amount of resolutions needed to obtain an answer. We believe it is useful, at least from the point of view of analyzing systems, to make a strict distinction between parallelism issues and computation-rule related issues. To this end, we define parallelism as the simultaneous execution of a number of *independent* sequences of resolutions, *taken from those which would have to be performed in any case as determined by the computation rules.* We call each such sequence a *thread* of execution. Note that as soon as there is an *actual* (i.e., run-time) dependency between two sequences, one has to wait for the other and therefore parallelism does not occur for some time. Thus, such sequences contain several threads. Exploiting parallelism means taking a fixed-size computation (determined by the computation rules), splitting it into independent threads related by dependencies (building a dependency graph), and assigning these segments to different agents. Both the partitioning and the agent assignment can be performed statically or dynamically. The objective of parallelism in this definition is simply to *perform the same amount of work in less time.*

We consider as an example a typical or-parallel system. Let us assume a finite tree, with no cuts or side-effects, and that all solutions are required. In a first approximation we could consider that the computation rules in such a system are the same as in Prolog and thus the same tree is explored and the number of resolution steps is the same. Exploiting (or-)parallelism then means taking branches of the resolution tree (which have no dependencies, given the assumptions) and giving them to different agents. The result is a performance gain that is independent of any performance implications of the computation rule. As is well known, however, if only (any) one solution is needed, then such a system can behave quite differently from Prolog: if the leftmost solution (the one Prolog would find) is deep in the tree, and there is another, shallower solution to its right, the or-parallel system may find this other solution first. Furthermore, it may do this after having explored a different portion of the tree which is potentially smaller (although also potentially bigger). The interesting thing to realize from our point of view is that part of the possible performance gain (which sometimes produces "super-linear" speedups) comes in a fundamental way from a change in the computation rule, rather than from parallel execution itself. It is not due to the fact that several agents are operating but to the different way in which the tree is being explored ("more breath-first").[1]

A similar phenomenon appears for example in independent and-parallel systems if they incorporate a certain amount of "intelligent failure": computation may be saved. We would like this to be seen as associated to a smarter computation rule that is taking advantage of the knowledge of the independence of some goals rather than having really anything to do with the parallelism. In contrast, also the possibility of performing additional work arises: unless non-failure can be proved ahead of time, and-parallel systems necessarily need to be speculative to a certain degree in order to obtain speedups. However such speculation can in fact be controlled so that no slow down occurs [18].

Another interesting example to consider is the Andorra-I system. The basic Andorra principle underlying this system states (informally) that deterministic reductions are performed ahead of time and possibly in parallel. This principle would be seen from our point of view as actually two principles, one related to the computation rules and another to parallelism. From the computation rule point of view the bottom line is that deterministic reductions are executed first. This is potentially very useful in practice since it can result in a change (generally a reduction, although the converse may also be true) of the number of resolutions needed to find a solution. Once the computation rule is isolated the remaining part of the rule is related to parallelism and can be seen simply as stating that deterministic reductions can be executed in parallel. Thus, the "parallelism part" of the basic Andorra principle, once isolated from the computation rule part, brings a basic principle to parallelism: that of the general convenience of parallel execution of deterministic threads.

We believe that the separation of computation rule and parallelism issues mentioned above allows enlarging the applicability of the interesting principles brought in by many current models.

2.2 Abstracting Away the Granularity Level: The Fundamental Principles

Having argued for the separation of parallelism issues from those that are related to computation rules, we now concentrate on the fundamental principles governing parallelism in the different models proposed. We argue that moving a principle from one system to another can often be done quite easily if another such "separation" is performed: isolating the principle itself from the *level of granularity* at which it is applied. This means viewing the parallelizing principle involved as associated to a generic concept of thread, to be particularized for each system, according to the fundamental unit of parallelism used in such system.

As an example, and following these ideas, the fundamental principle of determinism used in the basic Andorra model can be applied to the &-Prolog system. The basic unit of parallelism considered when parallelizing programs in the classical &-Prolog tools is the subtree corresponding to the complete resolution of a given goal in the resolvent. If the basic Andorra principle is applied at this level of granularity its implications are that deterministic subtrees can and should be executed in parallel (even if they are "dependent" in the classical sense). Moving the notions of determinism in the other direction, i.e. towards a finer level of granularity, one can think of applying the principle at the level of bindings, rather than clauses, which yields the concept of "binding determinism" of PNU-Prolog [30].

In fact, the converse can also be done: the underlying principles of &-Prolog w.r.t. parallelism –basically its independence rules– can in fact be applied at the granularity level of the Andorra model. The concept of independence in the context of &-Prolog is defined informally as requiring that a

[1]This can be observed for example by starting a Muse or an Aurora system with several "workers" on a uniprocessor machine. In this experiment it is possible sometimes to obtain a performance gain w.r.t. a sequential Prolog system even though there is no parallelism involved – just a *coroutining* computation rule, in this case implemented by the multitasking operating system.

part of the execution "will not be affected" by another. Sufficient conditions –strict and non-strict independence [18]– are then defined which are shown to ensure this property. We argue that applying these concepts at the granularity level of the Andorra model gives some new ways of understanding the model and some new solutions for its parallelization. In order to do this it is quite convenient to look at the basic operations in the light of David Warren's *extended* Andorra model.[2] The extended Andorra model brings in the first place the idea of presenting the execution of logic programs as a series of simple, low level operations on and-or trees. In addition to defining a lower level of granularity, the extended Andorra model incorporates some principles which are related in part to parallelism and in part to computation rule related issues such as the above mentioned basic Andorra principle and the avoidance of re-computation of goals.

On the other hand the extended Andorra model also leaves several other issues relatively more open. One example is that of when nondeterministic reductions may take place in parallel. One answer for this important and relatively open issue was given in the instantiation of the model in the AKL language. In AKL the concept of "stability" is defined as follows: a configuration (partial resolvent) is said to be stable if it cannot be affected by other sibling configurations. In that case the operational semantics of AKL allow the non-determinate promotion to proceed. Note that the definition is, not surprisingly, equivalent to that of independence, although applied at a different granularity level. Unfortunately stability/independence is in general an undecidable property. However, applying the work developed in the context of independent and-parallelism at this level of granularity provides sufficient conditions for it. The usefulness of this is underlined by the fact that the current version of AKL incorporates the relatively simple notion of strict independence (i.e. the absence of variable sharing) as its stability rule. However, the presentation above clearly marks the way for incorporating more advanced concepts, such as non-strict independence, as a sufficient condition for the independence/stability rule. As will be mentioned, we are actively working on compile-time detection of non-strict independence, which we believe will be instrumental in this context. Furthermore, and as we will show, when adding constraint support to a system the traditional notions of independence are no longer valid and both new definitions of independence and sufficient conditions for it need to be developed. We believe that the view proposed herein allows the direct application of general results concerning independence in constraint systems to several realms, such as the extended Andorra model and AKL.

Another way of moving the concept of independence to a finer level of granularity is to apply it at the binding level. This yields a rule which states that dependent bindings of variables should wait for their leftmost occurrences to complete (in the same way as subtrees wait for dependent subtrees to their left to complete in the standard independent and-parallelism model), which is essentially the underlying rule of the DDAS model [32]. In fact, one can imagine applying the principle of non-strict independence at the level of bindings, which would yield a "non-strict" version of DDAS which would not require dependent bindings to wait for bindings to their left which are guaranteed to never occur, or for bindings which are guaranteed to be compatible with them.

With this view in mind we argue that there are essentially four fundamental principles which govern exploitation of parallelism:

- *independence*, which allows parallelism among non-deterministic threads,
- *determinacy*, which allows parallelism among dependent threads,
- *non-failure*, which allows guaranteeing non-speculativeness, and
- *granularity*, which allows guaranteeing speedup in the presence of overheads.

[2]This is understandable, given that adding independent and-parallelism to the basic Andorra model was one of the objectives in the development of its extended version.

2.3 User-level Concurrency

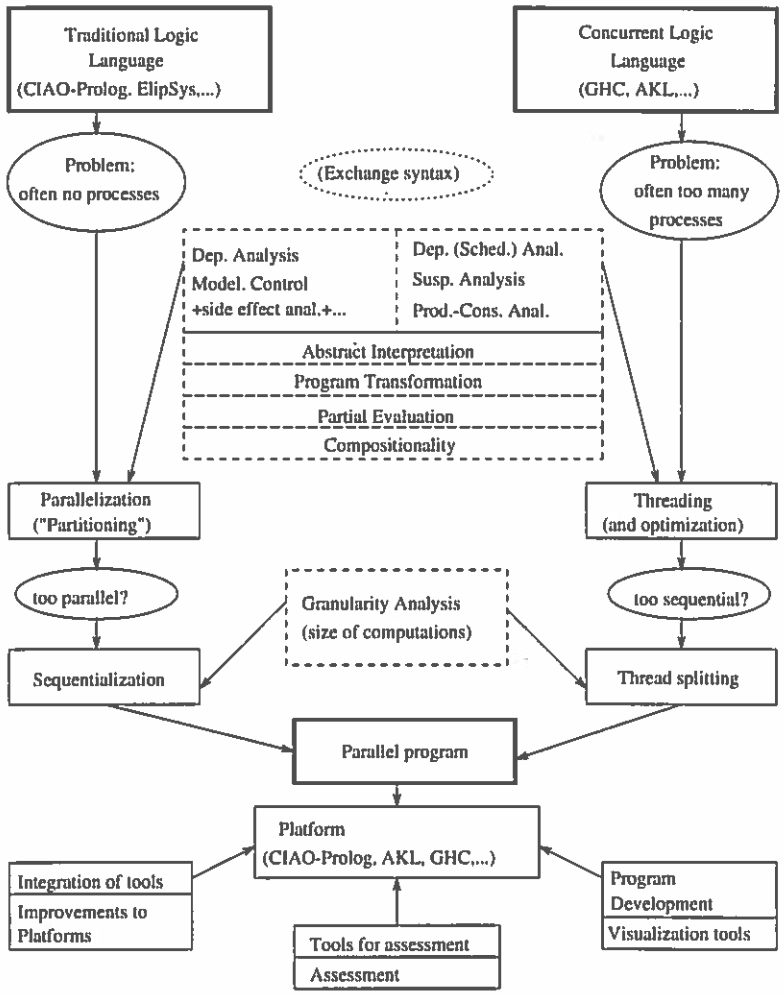
Similarly to the separations mentioned above (parallelism vs. computation rule and principles vs. granularity level of their application) we also believe in a separation of "concurrency" from both parallelism and computation rules. We believe concurrency is most useful when it is explicitly controlled by the user and should be separate from the implicit computation rules. This is in contrast with parallelism, which ideally should be transparent to the user, and with smart computation rules of which the user should only be aware in the sense of being able to derive an upper bound on the amount of computation involved in running a program for a given query using that rule. Space limitations prevent us from elaborating more on this topic or that of the separation between concurrency and parallelism. However, an example of an application of the latter can be seen in *schedule analysis*, where the maximal essential components of concurrency are isolated and sequenced to allow the most efficient possible execution of the concurrent program by one agent [21]. Schedule analysis is, after all, an application of the concept of dependence (or, conversely, independence) at a certain level of granularity in order to "unparallelize" a program, and is thus based on the same principles as automatic parallelization.

2.4 Towards a General-Purpose Implementation

We believe that the points regarding the separation of issues and fundamental principles sketched in the previous sections at the same time explain and are supported by the recent trend towards convergence in the implemenation techniques of systems that are in principle very different, such as the various parallel implementations of Prolog on one hand (see, for example, [17, 27, 2]) and the implementations of the various committed choice languages on the other (see, for example, [7, 8, 14, 19, 24, 35, 38, 39]). The former are based on schemes for parallelizing a sequential language; they tend to be stack-based, in the sense that (virtual) processors allocate environments on a stack and execute computations "locally" as far as possible until there is no more work to do, at which point they "steal" work from a busy processor. The latter, by contrast, are based on concurrent languages with dataflow synchronization; they tend to be heap-based, in the sense that environments are generally allocated on a heap, and there is (at least conceptually) a shared queue of active tasks.

The aforementioned convergence can be observed in that, on one hand, driven by the demonstrated utility of delay primitives in sequential Prolog systems (e.g., the `freeze` and `block` declarations of Sicstus Prolog [6], `when` declarations of NU-Prolog [36], etc.), parallel Prolog systems have been incorporating capabilities to deal with user-defined suspension and coroutining behaviors—for example, &-Prolog allows programmer-supplied *wait*-declarations, which can be used to express arbitrary control dependencies. In sequential Prolog systems with delay primitives, delayed goals are typically represented via heap-allocated "suspension records," and such goals are awakened when the variables they are suspended on get bindings [5]. Parallel Prolog systems inherit this architecture, leading to implementations where individual tasks are stack-oriented, together with support for heap-allocated suspensions and dataflow synchronization. On the other hand, driven by a growing consensus that some form of "sequentialization" is necessary to reduce the overhead of managing fine-grained parallel tasks on stock hardware (see, for example, [13, 37, 22]), implementors of committed choice languages are investigating the use of compile-time analyses to coalesce fine-grained tasks into coarser-grained sequential threads that can be implemented more efficiently. This, again, leads to implementations where individual sequential threads execute in a stack-oriented manner, but where sets of such threads are represented via heap-allocated activation records that employ dataflow synchronization. Interestingly, and conversely, in the context of parallel Prolog systems, there is also a growing body of work trying to address the problem of automatic parallelizing compilers often "parallelizing too much" which appears if the target architecture is not capable of supporting fine grain parallelism. Figure

2.4 illustrates this (and in fact reflects the interactions among the partners of the ParForCE Esprit project, where some of these interactions are being investigated).



This convergence of trends is exciting: it suggests that we are beginning to understand the essential implementation issues for these languages, and that from an implementor's perspective these languages are not as fundamentally different as was originally believed. It also opens up the possibility of having a general purpose abstract machine to serve as a compilation target for a variety of languages. As mentioned before this is precisely one of the objectives of the CIAO system. Encouraging initial results in this direction have been demonstrated in the sequential context by the QD-Janus system [12] of S. Debray and his group. QD-Janus, which compiles down to Sicstus Prolog and uses the delay primitives of the Prolog system to implement dataflow synchronization, turns out to be more than three times faster, on the average, than Kliger's customized implementation of FCP(:) [23] and requires two orders of magnitude less heap memory [11]. We believe that this point will also extend to parallel systems: as noted above, the &-Prolog system already supports stack-oriented parallel execution together with arbitrary control dependencies, suspension, and dataflow synchronization via user-supplied *wait*-declarations, all characteristics that CIAO inherits. This suggests that the dependence graphs and *wait*-declarations of &-Prolog/CIAO can serve as a common intermediate language, and its runtime system can act as an appropriate common low-level implementation, for a variety of parallel logic programming implementations. We do not mean to suggest that the performance of such a system will be *optimal* for all possible logic programming languages: our claim is rather that it will provide a way to researchers in the community implement their languages with considerably less effort than has been possible to date, and yet attain reasonably good performance. We are currently exploring these points in collaboration with S. Debray.

3 Some of our recent work in this context

We now provide an overview of our recent work in filling some of the gaps that, in our understanding, are missing in order to fulfill the objectives outlined in the previous section.

3.1 Parallelism based on Non-Strict Independence

One of our starting steps is to improve the independence-based detection of parallelism based on information that can be obtained from global analysis using the current state of the art in abstract interpretation. We have had a quite successful experience using this technique for detecting the classical notion of "strict" independence. These results are summarized in [3], which compares the performance of several abstract interpretation domains and parallelization algorithms using the &-Prolog compiler and system.

While these results are quite encouraging there is another notion of independence – "non-strict" independence [18] – which ensures the same important "no slow down" properties than the traditional notion of strict independence and allows considerable more parallelism than strict independence [33]. The support of non-strict independence requires, however, a review of our compile-time parallelization technology which to date has been exclusively based on strict independence. In [4] we describe some of our recent work filling this gap. Rules and algorithms are provided for detecting and annotating non-strict independence at compile-time. We also propose algorithms for combined compile-time/run-time detection, including run-time checks for this type of parallelism, which in some cases turn out to be different from the traditional groundness and independence checks used for strict independence. The approach is based on the knowledge of certain properties about run-time instantiations of program variables —sharing, groundness, freeness, etc.— for which compile-time technology is available, with new approaches being currently proposed. Rather than dealing with the analysis itself, we present how the analysis results can be used to parallelize programs.

3.2 Parallelization in the Presence of Constraints: Independence / Stability

In the CIAO-Prolog system, from the language point of view, we assume a constraint-based, non-deterministic logic programming language. As such, and apart from the concurrency/coroutining primitives, the user language can be viewed as similar to Prolog when working on the Herbrand domain, and to systems such as CLP(R) or CHIP when working over other domains. This implies that the traditional notions of independence / stability need to be evaluated in this context and, if necessary, extended to deal with the fact that constraint solving is occurring in the actual execution in lieu of unification.

Previous work in the context of traditional Logic Programming languages has concentrated on defining independence in terms of preservation of search space, and such preservation has then been achieved by ensuring that either the goals do not share variables (*strict independence*) or if they share variables, that they do not "compete" for their bindings (*non-strict independence*).

In [10] we have shown (in collaboration with Monash University) that a naive extrapolation of the traditional notions of independence to Constraint Logic Programming is unsatisfactory (in fact, wrong) for two reasons. First, because interaction between variables through constraints is more complex than in the case of logic programming. Second, in order to ensure the efficiency of several optimizations not only must independence of the search space be considered, but also an orthogonal issue – "independence of constraint solving." We clarify these issues by proposing various types of search independence and constraint solver independence, and show how they can be combined to allow different independence-related optimizations, in particular parallelism. Sufficient conditions for independence which can be evaluated "a-priori" at run-time and are easier to identify at compile-time than the original definitions, are also proposed. Also, it has been shown how the concepts proposed, when applied to traditional Logic Programming, render the traditional notions and are thus a strict generalization of such notions.

3.3 Extending Global Analysis Technology to CLP

As mentioned before, since many optimizations, including independence / stability detection, are greatly aided by (and sometimes even require) global analysis, traditional global analysis techniques have to be extended to deal with the fact that constraint solving is occurring in the actual execution in lieu of unification. In [9] we present and illustrate with an implementation a practical approach to the dataflow analysis of programs written in constraint logic programming (CLP) languages using abstract interpretation. We argue that, from the framework point of view, it suffices to propose quite simple extensions to traditional analysis methods which have already been proved useful and practical and for which efficient fixpoint algorithms have been developed. This is shown by proposing a simple but quite general extension to the analysis of CLP programs of Bruynooghe's traditional framework, and describing its implementation – the "PLAI" system. As the original, the framework is parametric and we provide correctness conditions to be met by the abstract domain related functions to be provided. In this extension constraints are viewed not as "suspended goals" but rather as new information in the store, following the traditional view of CLP. Using this approach, and as an example of its use, a complete, constraint system independent, abstract analysis is presented for approximating definiteness information. The analysis is in fact of quite general applicability. It has been implemented and used in the analysis of CLP(R) and Prolog-III applications. Results from this implementation are also presented which show good efficiency and accuracy for the analysis.

This framework, combined with the ideas of [10] (and [29]) presented in the previous section, is the basis for our current development of automatic parallelization tools for CLP programs, and, in particular, of the parallelizer for the CIAO-Prolog system.

3.4 Extending Global Analysis Technology for Explicit Concurrency

Another step that has to be taken in adapting current compile-time technology to CIAO systems is to develop global analysis technology which can deal with the fact that the new computation rules allow the specification of concurrent executions. While there have been many approaches proposed in the literature to address this problem, in a first approach we focus on a class of languages (which includes modern Prologs with delay declarations) which provide both sequential and concurrent operators for composing goals. In this approach we concentrate on extending traditional abstract interpretation based global analysis techniques to incorporate these new computation rules. This gives a practical method for analyzing (constraint) logic programming languages with (explicit) dynamic scheduling policies, which is at the same time equally powerful as the older methods for traditional programs.

We have developed, in collaboration with the University of Melbourne, a framework for global dataflow analysis of this class of languages [28]. First, we give a denotational semantics for languages with dynamic scheduling which provides the semantic basis for our generic analysis. The main difference with denotational definitions for traditional Prolog is that sequences of delayed atoms must also be abstracted and are included in "calls" and "answers." Second, we give a generic global dataflow analysis algorithm which is based on the denotational semantics. Correctness is formalized in terms of abstract interpretation. The analysis gives information about call arguments and the delayed calls, as well as implicit information about possible call schedulings at runtime. The analysis is generic in the sense that it has a parametric domain and various parametric functions. Finally, we demonstrate the utility and practical importance of the dataflow analysis algorithm by presenting and implementing an example instantiation of the generic analysis which gives information about groundness and freeness of variables in the delayed and actual calls. Some preliminary test results are included in which the information provided the implemented analyzer is used to reduce the overhead of dynamic scheduling by removing unnecessary tests for delaying and awakening, to reorder goals so that atoms are not delayed, and to recognize calls which are "independent" and so allow the program to be run in parallel.

3.5 Granularity Analysis

While logic programming languages offer a great deal of scope for parallelism, there is usually some overhead associated with the execution of goals in parallel because of the work involved in task creation and scheduling. In practice, therefore, the "granularity" of a goal, i.e. an estimate of the work available under it, should be taken into account when deciding whether or not to execute a goal in parallel as a separate task. Building on the ideas first proposed in [13] we describe in [25] a proposal for an automatic granularity control system, which is based on an accurate granularity analysis and program transformation techniques. The proposal covers granualrity control of both and-parallelism and or-parallelism. The system estimates the granularities of goals at compile time, but they are actually evaluated at runtime. The runtime overhead associated with our approach is usually quite small, and the performance improvements resulting from the incorporation of grain size control can be quite good. Moreover a static analysis of the overhead associated with granularity control process is performed in order to decide its convenience.

The method proposed requires among other things knowing the size of the terms to which program variables are bound at run-time (something which is useful in a class of optimizations which also include recursion elimination). Such size is difficult to even approximate at compile time and is thus generally computed at run-time by using (possibly predefined) predicates which traverse the terms involved. In [26] we propose a technique based on program transformation which has the potential of performing this computation much more efficiently. The technique is based on finding program procedures which are called before those in which knowledge regarding term sizes is needed and which traverse the terms whose size is to be determined, and transforming such procedures so that they compute term sizes "on the fly". We present a systematic way of determining whether a given program can be transformed in order to compute a given term size at a given program point without additional term traversal. Also, if several such transformations are possible our approach allows finding minimal transformations under certain criteria. We also discuss the advantages and applications of our technique and present some performance results.

3.6 Memory Management and Scheduling in Non-deterministic And-parallel Systems

From our experience with the &-Prolog system implementation [17], the results from the DDAS simulator [32], and from informal conversations with the Andorra-I developers, efficient memory management in systems which exploit and-parallelism is a problem for which current solutions are not completely satisfactory. This appears to be specially the case with and-parallel systems which support don't-know nondeterminism or deep guards. We believe *non-deterministic* and-parallel schemes to be highly interesting in that they present a relatively general set of problems to be solved (including most of those encountered in the memory management of or-parallel only systems) and have chosen to concentrate on their study.

In collaboration with U. of Bristol, we have developed a distributed stack memory management model which allows flexible scheduling of goals. Previously proposed models are lacking in that they impose restrictions on the selection of goals to be executed or they may require a large amount of virtual memory. Our measurements imply that the above mentioned shortcomings can have significant performance impacts, and that the extension that we propose of the "Marker Model" allows flexible scheduling of goals while keeping (virtual) memory consumption down. We also discuss methods for handling forward and backward execution, cut, and roll back. Also, we show that the mechanism proposed for flexible scheduling can be applied to the efficient handling of the very general form of suspension that can occur in systems which combine several types of non-deterministic and-parallelism and advanced computation rules, such as PNU-Prolog [30], IDIOM [16], DDAS [32], AKL [20], and, in general, those that can be seen as an instantiation of the extended Andorra model [40]. Thus, we believe that the results may be applicable to a whole class of and- and or-parallel systems. Our solutions and results are described more fully in [34].

3.7 Incorporating Or-Parallelism: The ACE Approach

Another important issue is the incorporation of Or-parallelism to an and-parallel system. This implies well known problems related to or-parallelism itself, such as the maintenance of several binding environments, as well as new problems such as the interactions of the multiplicity of binding environments and threads of or-parallel computation with the scoping and memory management requirements of and-parallelism. The stack copying approach, as exemplified by the MUSE system, has been shown to be a quite successful alternative for representing multiple environments during or-parallel execution of logic programs. In collaboration with the U. of New Mexico and U. of Bristol we have developed an approach for parallel implementation of logic programs, described more fully in [15], which we believe is capable of exploiting both or-parallelism and independent and-parallelism (as well as other types of and-parallelism) in an efficient way using stack copying ideas. This model combines such ideas with proven techniques in the implementation of independent and-parallelism, such as those used in &-Prolog. We show how all solutions to non-deterministic and-parallel goals are found without repetitions. This is done through re-computation as in Prolog (and &-Prolog), i.e., solutions of and-parallel goals are not shared. We propose a scheme for the efficient management of the address space in a way that is compatible with the apparently incompatible requirements of both and- and or-parallelism. This scheme allows incorporating and combining the memory management techniques used in (non-deterministic) and-parallel systems, such as those mentioned in the previous section, and memory management techniques of or-parallel systems, such as incremental copying. We also show how the full Prolog language, with all its extra-logical features, can be supported in our and-or parallel system so that its sequential semantics is preserved. The resulting system retains the advantages of both purely or-parallel systems as well as purely and-parallel systems. The stack copying scheme together with our proposed memory management scheme can also be used to implement models that combine dependent and-parallelism and or-parallelism, such as Andorra and Prometheus.

References

[1] K.A.M. Ali and R. Karlsson. The Muse Or-Parallel Prolog Model and its Performance. In *1990 North American Conference on Logic Programming*. MIT Press, October 1990.

[2] P. Brand, S. Haridi, and D.H.D. Warren. Andorra Prolog—The Language and Application in Distributed Simulation. In *International Conference on Fifth Generation Computer Systems*. Tokyo, November 1988.

[3] F. Bueno, M. García de la Banda, and M. Hermenegildo. Effectiveness of Global Analysis in Strict Independence-Based Automatic Program Parallelization. Technical Report TR Number CLIP7/93.0, T.U. of Madrid (UPM), Facultad Informática UPM, 28660-Boadilla del Monte, Madrid-Spain, October 1993.

[4] D. Cabeza and M. Hermenegildo. Towards Extracting Non-strict Independent And-parallelism Using Sharing and Freeness Information. Technical Report TR Number CLIP5/92.1, U. of Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, August 1993.

[5] M. Carlsson. Freeze, Indexing, and Other Implementation Issues in the Wam. In *Fourth International Conference on Logic Programming*, pages 40–58. University of Melbourne, MIT Press, May 1987.

[6] M. Carlsson. *Sicstus Prolog User's Manual.* Po Box 1263, S-16313 Spanga, Sweden, February 1988.

[7] K. L. Clark and S. Gregory. Notes on the Implementation of Parlog. *Journal of Logic Programming*, 2(1), April 1985.

[8] Jim Crammond. Scheduling and Variable Assignment in the Parallel Parlog Implementation. In *1990 North American Conference on Logic Programming.* MIT Press, 1990.

[9] M.J.García de la Banda and M. Hermenegildo. A Practical Approach to the Global Analysis of Constraint Logic Programs. In *1993 International Logic Programming Symposium.* MIT Press, Cambridge, MA, October 1993.

[10] M.J.García de la Banda, M. Hermenegildo, and K. Marriott. Independence in Constraint Logic Programs. In *1993 International Logic Programming Symposium.* MIT Press, Cambridge, MA, October 1993.

[11] S. K. Debray. Implementing logic programming systems: The quiche-eating approach. In *ICLP '93 Workshop on Practical Implementations and Systems Experience in Logic Programming*, Budapest, Hungary, June 1993.

[12] S. K. Debray. Qd-janus : A sequential implementation of janus in prolog. *Software—Practice and Experience*, 23(12):1337–1360, December 1993.

[13] S. K. Debray, N.-W. Lin, and M. Hermenegildo. Task Granularity Analysis in Logic Programs. In *Proc. of the 1990 ACM Conf. on Programming Language Design and Implementation*, pages 174–188. ACM Press, June 1990.

[14] I. Foster and S. Taylor. *Strand* : A practical parallel programming tool. In *1989 North American Conference on Logic Programming*, pages 497–512. MIT Press, October 1989.

[15] G. Gupta, M. Hermenegildo, Enrico Pontelli, and Vítor Santos Costa. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *International Conference on Logic Programming.* MIT Press, June 1994. to appear.

[16] G. Gupta, V. Santos-Costa, R. Yang, and M. Hermenegildo. IDIOM: Integrating Dependent and-, Independent and-, and Or-parallelism. In *1991 International Logic Programming Symposium*, pages 152–166. MIT Press, October 1991.

[17] M. Hermenegildo and K. Greene. The &-prolog System: Exploiting Independent And-Parallelism. *New Generation Computing*, 9(3,4):233–257, 1991.

[18] M. Hermenegildo and F. Rossi. Strict and Non-Strict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions. *Journal of Logic Programming*, 1993. To appear.

[19] A. Houri and E. Shapiro. A sequential abstract machine for flat concurrent prolog. Technical Report CS86-20, Dept. of Computer Science, The Weizmann Institute of Science, Rehovot 76100, Israel, July 1986.

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

[21] A. King and P. Soper. Reducing scheduling overheads for concurrent logic programs. In *International Workshop on Processing Declarative Knowledge*, Kaiserslautern, Germany, (1991). Springer-Verlag.

[22] Andy King and Paul Soper. Schedule Analysis of Concurrent Logic Programs. In Krzysztof Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 478–492, Washington, USA, 1992. The MIT Press.

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

[24] M. Korsloot and E. Tick. Compilation techniques for nondeterminate flat concurrent logic programming languages. In *1991 International Conference on Logic Programming*, pages 457–471. MIT Press, June 1991.

[25] P. López and M. Hermenegildo. An automatic sequentializer based on program transformation. Technical report, T.U. of Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, April 1993.

[26] P. López and M. Hermenegildo. Dynamic Term Size Computation in Logic Programs via Program Transformation. Technical report, T.U. of Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, April 1993. Presented at the 1993 COMPULOG Area Meeting on Parallelism and Implementation Technologies.

[27] E. Lusk et. al. The Aurora Or-Parallel Prolog System. *New Generation Computing*, 7(2,3), 1990.

[28] K. Marriott, M. Garcia de la Banda, and M. Hermenegildo. Analyzing Logic Programs with Dynamic Scheduling. In *Proceedings of the 20th. Annual ACM Conf. on Principles of Programming Languages*. ACM, January 1994.

[29] U. Montanari, F. Rossi, F. Bueno, M. García de la Banda, and M. Hermenegildo. Contextual Nets and Constraint Logic Programming: Towards a True Concurrent Semantics for CLP. Technical report, T.U. of Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, January 1993. To be Presented at the ICLP'93 Post Conference Workshop on Concurrent Constraint Logic Programming.

[30] L. Naish. Parallelizing NU-Prolog. In *Fifth International Conference and Symposium on Logic Programming*, pages 1546–1564. University of Washington, MIT Press, August 1988.

[31] V. Santos-Costa, D.H.D. Warren, and R. Yang. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-parallelism. In *Proceedings of the 3rd. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, April 1990.

[32] K. Shen. Exploiting Dependent And-Parallelism in Prolog: The Dynamic, Dependent And-Parallel Scheme. In *Proc. Joint Int'l. Conf. and Symp. on Logic Prog.* MIT Press, 1992.

[33] K. Shen and M. Hermenegildo. A Simulation Study of Or- and Independent And-parallelism. In *1991 International Logic Programming Symposium*. MIT Press, October 1991.

[34] K. Shen and M. Hermenegildo. A Flexible Scheduling and Memory Management Scheme for Non-Deterministic, And-parallel Execution of Logic Programs. Technical report, T.U. of Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, April 1993. Presented at the ICLP'93 Post Conference Workshop on Logic Program Implementation.

[35] S. Taylor, S. Safra, and E. Shapiro. A parallel implementation of flat concurrent prolog. In E.Y. Shapiro, editor, *Concurrent Prolog: Collected Papers*, pages 575–604, Cambridge MA, 1987. MIT Press.

[36] J. Thom and J. Zobel. *NU-Prolog Reference Manual*. Dept. of Computer Science, U. of Melbourne, May 1987.

[37] E. Tick. Compile-Time Granularity Analysis of Parallel Logic Programming Languages. In *International Conference on Fifth Generation Computer Systems*. Tokyo, November 1988.

[38] E. Tick and C. Bannerjee. Performance evaluation of monaco compiler and runtime kernel. In *1993 International Conference on Logic Programming*, pages 757–773. MIT Press, June 1993.

[39] K. Ueda. Guarded Horn Clauses. In E.Y. Shapiro, editor, *Concurrent Prolog: Collected Papers*, pages 140–156. MIT Press, Cambridge MA, 1987.

[40] D.H.D. Warren. The Extended Andorra Model with Implicit Control. In Sverker Jansson, editor, *Parallel Logic Programming Workshop*, Box 1263, S-163 13 Spanga, SWEDEN, June 1990. SICS.

Using Run-Time List Sizes to Guide Parallel-Thread Creation

Lorenz Huelsbergen
AT&T Bell Laboratories
lorenz@research.att.com

James R. Larus
University of Wisconsin–Madison
larus@cs.wisc.edu

Alexander Aiken
University of California–Berkeley
aiken@cs.berkeley.edu

1 Introduction

Functional languages do not overly constrain a program's evaluation order with data dependences. This simplifies automatic parallelization: multiple arguments in a strict function application can evaluate in parallel, for example. Abundant parallelism, however, does not directly lead to effective parallel implementations. Efficient implementation of a functional language on a parallel architecture remains difficult in part because the creation of a parallel thread incurs considerable overhead costs [10, 16, 18, 15].

For an implementation to be efficient, it must decide which parallelism in a program is beneficial; that is, whether parallel evaluation of a given expression will speed program execution. If an expression contains less computation than the cost of creating a thread for the expression, parallel evaluation of the thread will slow program execution. Figure 1 shows the effect that scheduling overheads can have on overall execution times.

In this paper, we present a new technique, *dynamic granularity estimation* (**dge**), that uses the run-time sizes of data structures to create parallel threads only when they are known to be beneficial. This technique is based on the observation that a function's time complexity often depends on the size of the dynamic data with which it computes. For simplicity, we describe **dge** for lists—the general scheme can, however, be applied to programs that manipulate other data structures (*e.g.*, trees, DAGs, and arrays).

In a list-based language, **dge** conservatively determines, for a program function f applied to a list parameter l, the lengths of l for which the cost of computing the application $e \equiv (f\ l)$ always exceeds the overhead of creating a thread for e's concurrent evaluation. Initial empirical evidence, gathered in an implementation of **dge** in Standard ML of New Jersey (SML/NJ) [2] on a parallel shared-memory machine, suggests that the run-time costs of **dge** are small and that **dge** can substantially reduce a program's parallel execution time.

Dynamic granularity estimation is a hybrid; it is composed of dynamic and static components (*cf.*. [13, 12]). Hybrid techniques are necessary for language parallelization since purely-static analyses are fundamentally limited. Static analysis for **dge** is in the form of an *abstract interpretation* [4, 1] that identifies functions whose time complexity is dependent on the list data structures passed to them as parameters. The compiler statically identifies program points at which the length of a list always influences the cost of an application expression. When evaluation reaches such a point, compiler-inserted code consults an approximation to the list's length (maintained dynamically) to determine whether it is beneficial to evaluate an application as a separate parallel thread. The dynamic component of **dge** approximates list lengths at run time.

The quicksort function (qs) of Figure 2 provides an example. In qs, the arguments to append can evaluate in parallel. Parallel evaluation of these expressions is advantageous if the costs of the recursive applications of qs exceed the cost of creating and scheduling them as parallel threads. However, when the length of a sublist (l or g) is small (*e.g.*, zero), creating a parallel thread to sort the sublist is counterproductive. In this case, the arguments to append should evaluate

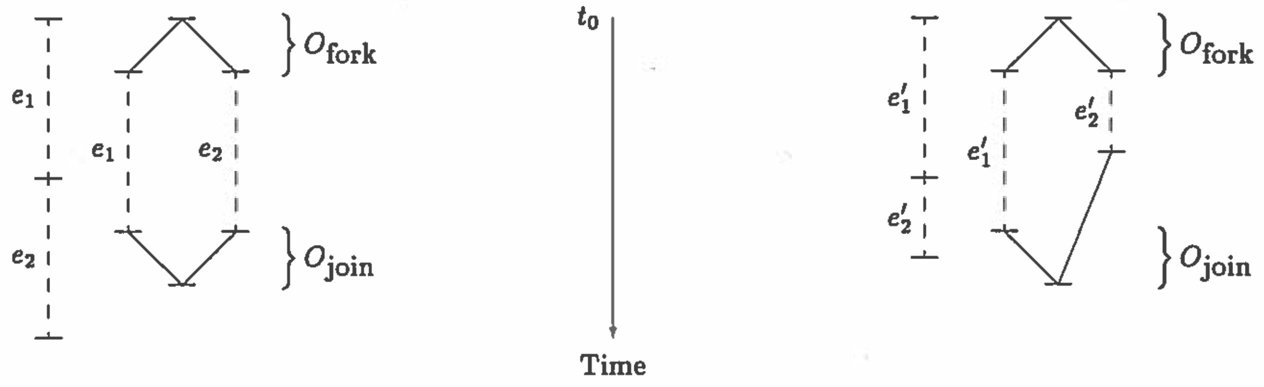


Figure 1: The impact of overhead. Time starts at t_0. The concurrent evaluation of e_1 and e_2, with overhead ($O = O_{\text{fork}} + O_{\text{join}}$) taken into account, completes before their sequential evaluation and is, therefore, beneficial. Concurrent evaluation of e'_1 and e'_2, however, slows the program's evaluation since e'_2 does not contain enough computation to offset scheduling overheads.

```
fun qs p [] = []
  | qs p (x::xs) =
        let fun split l =
                let fun split' [] less greater = (less,greater)
                      | split' (y::ys) less greater =
                            if (p y x) then
                                split' ys (y::less) greater
                            else
                                split' ys less (y::greater)
                in
                    split' l [] []
                end
            val (l,g) = split xs
        in
            if (l̄ > cutoff) andalso (ḡ > cutoff) then
                append|| (qs p l) (x::(qs p g))
            else
                append (qs p l) (x::(qs p g))
        end
```

Figure 2: Functional quicksort automatically restructured by dynamic granularity estimation. Static analysis determines that the amounts of computation in the arguments to append depend on the lengths (denoted $\bar{\texttt{l}}$ and $\bar{\texttt{g}}$) of the sublists produced by `split`. The compiler inserts a run-time check (the conditional in `qs`'s body) to examine the lengths of `l` and `r` (stored with the list representation). Based on these dynamic lengths, the check decides whether to create parallel threads (append$_{||}$ evaluates its arguments in parallel). The compiler also deduces the `cutoff` value.

sequentially. The static analysis of **dge** identifies list lengths for which the cost of applying `qs` to a list of that length is always greater than the overhead incurred in creating a new thread for the application's concurrent evaluation. At run time, **dge** approximates the lengths of all lists. The length information of the lists bound to the identifiers `l` and `g` in the `qs` function are available for making the final parallelization decision.

Dynamic techniques, like **dge**, that examine the sizes of data structures to conditionally select parallel evaluation are necessary since compile-time expression scheduling is fundamentally limited. This is evident from the `qs` example. When a statically-unknown list reaches `qs`, the sublist partition that `qs`'s auxiliary `split` function creates is also unknown. Therefore, the costs of the recursive applications of `qs` that sort the sublists cannot be known at compile time. In the absence of precise static information about `qs`'s list parameter, it is therefore not possible to decide statically when it is advantageous to concurrently evaluate `qs`'s recursive applications.

In languages with explicit constructs for thread creation and synchronization, programmers typically use *cutoff* values to curb parallelism and to ensure that the program only creates large threads [7]. In the `qs` example, the programmer might explicitly check if the sublist being sorted contains $> k$ elements for some small k before creating parallel threads for append's arguments. Code remains portable with **dge** since the language's implementation—not the programmer—matches a cutoff to the underlying parallel architecture. The granularity of parallel threads is less of a programming issue when thread sizes are determined automatically.

In the next section, we describe the language under consideration for dynamic granularity estimation and introduce terminology. We then describe **dge**'s static (§3.1) and dynamic (§3.2) components, and discuss an initial implementation of this new technique (§4) and preliminary results (§5).

2 Preliminaries

The language under consideration for dynamic granularity estimation is the λ_v-calculus, a functional[1], call-by-value, higher-order language [19, 20]. The ground terms of λ_v are variables and constants; terms are expressions ($e \in \textsc{Exp}$) and values ($v \in \textsc{Val} \subset \textsc{Exp}$):

$$
\begin{array}{lcl@{\qquad\qquad}lcl@{\qquad\qquad}lcl}
x & \in & \textsc{Var} & e & ::= & v & v & ::= & b \\
b & \in & \textsc{Const} = \{\mathtt{nil}, \mathtt{true}, \mathtt{false}\} & & \mid & e\ e & & \mid & x \\
 & & & & \mid & \mathtt{if}\ e\ \mathtt{then}\ e\ \mathtt{else}\ e & & \mid & \lambda x.e \\
 & & & & \mid & \mathtt{cons}\ e\ e & & & \\
 & & & & \mid & \mathtt{hd}\ e & & & \\
 & & & & \mid & \mathtt{tl}\ e & & & \\
 & & & & \mid & \mathtt{isnull}\ e & & &
\end{array}
$$

We further assume that λ_v terms are well-typed.

For simplicity, we focus on the *list* as the dynamic structure for dynamic granularity estimation. This is because a list's size is simply its length. The syntax of λ_v therefore contains `cons`, `hd`, `tl`, and `isnull` directly.

Denote the time required to evaluate an expression e as $|e|$, the *cost* of e. The cost of a parallel thread to evaluate e is $|e|$ plus the overhead, O, required to create and schedule a parallel

[1]Restriction to a functional language allows efficient implementation of **dge**'s dynamic component that approximates the sizes of dynamic data at run time. In a functional language, a datum d's size can only monotonically increase whereas, in a language with assignment to reference values, d's size can decrease and the efficient propagation of d's new (reduced) size estimate to other data that share d is difficult.

thread.[2] Let $T \geq O$ be a machine-dependent cost *threshold* so that if $|e| > T$ then expression e is a candidate for parallel evaluation (*cf.* Figures 1 and 2). Costs are measured in integer *evaluation units* (e-units). An e-unit corresponds to—again for simplicity—the operational notion of function application [5]. For a given implementation, normalization of e-units is necessary since all function applications do not have identical costs (*e.g.*, functions may be compiled in line).

For λ_v, we assume that the evaluation of variables, constants and λ-abstractions incurs no cost (zero e-units) and that the evaluation of the other language terms costs one e-unit. Under these simplifying assumptions, for example, the application $(f\ (g\ l))$, where f and g are functions and l is a list, incurs a cost of at least two e-units (the applications of f and g each cost one), but complete evaluation of $(f\ (g\ l))$ may require many more e-units and may depend on the size (length) of l.

The length of list l is written as $\bar{l}$. When i is a natural number, $\bar{i}$ represents any list of length i.

3 The New Technique

Dynamic granularity estimation deduces at compile time whether a program function `f`'s complexity depends on the sizes of `f`'s list parameters. This information can then be used by a compiler to restructure an application $e \equiv$ `(f` l`)` by inserting a check of l's length that selects parallel evaluation of e only when e contains enough computation to warrant its parallel evaluation. The deduction of a function's evaluation cost relative to its list parameters and subsequent program restructuring (check insertion) constitute **dge**'s static component. The dynamic component of **dge** maintains lengths with lists at run time. This section first describes **dge**'s static component and then its dynamic component.

3.1 Static Component

The idea is to abstractly evaluate, at compile time, an application $e \equiv (f\ l)$ while counting the number of e-units required. The static e-unit count thus obtained is conservative; that is, static estimation of e-units does not overestimate the number of e-units that evaluation of an expression requires. For example, if static analysis of e indicates that $|e| = i$, then actual evaluation of e must require $\geq i$ e-units. Since the aim is to identify functions whose list parameters control their complexity, an abstract semantics that interprets a list l as its length, $\bar{l}$, is used. E-units are (conservatively) counted under this abstract semantics. We first give the standard semantics for the language and then the abstract semantics. To guarantee the termination of abstract evaluation, it is also necessary to bound the number of abstract evaluation steps (§3.1.3). This bound is naturally the threshold T (§2) at which parallel evaluation of a thread becomes beneficial (*i.e.*, overcomes scheduling overheads).

3.1.1 Standard Semantics $\mathcal{S}$

The dynamic objects of the standard semantics $\mathcal{S}$ are in Figure 3. Since the list is the dynamic structure of interest for granularity estimation, it is directly represented with dynamic objects rather than indirectly encoded in λ_v: The constant *nil* is the empty list and a *cons* pair $\langle v, l\rangle$ contains an element v and the list's tail l.

Figure 3 gives a standard semantics for the language. The operational style of the semantics is derived from Tofte's semantics [23]. The semantics given here, however, also contains *integer*

[2]It is assumed that the cost of creating and scheduling a thread is bounded and can be (empirically) determined for a given language implementation and machine architecture.

$$
\begin{array}{rcl}
b & \in & \textsc{Bool} = \{\mathit{true}, \mathit{false}\} \\
\langle v, l \rangle & \in & \textsc{Cons} = \textsc{DVal} \times \textsc{List} \\
l & \in & \textsc{List} = \{\mathit{nil}\} + \textsc{Cons} \\
[x, e, E] & \in & \textsc{Clos} = \textsc{Var} \times \textsc{Exp} \times \textsc{Env} \\
v & \in & \textsc{DVal} = \textsc{Bool} + \textsc{List} + \textsc{Clos} \\
E & \in & \textsc{Env} = \textsc{Var} \xrightarrow{fin} \textsc{DVal}
\end{array}
$$

$$\frac{x \mapsto v \in E}{E \vdash x \longrightarrow_0 v} \qquad \textbf{(var)}$$

$$\frac{}{E \vdash (\lambda x.e) \longrightarrow_0 [x, e, E]} \qquad \textbf{(abs)}$$

$$\frac{\begin{array}{r} E \ \vdash \ e_1 \longrightarrow_a [x, e, E'] \\ E \ \vdash \ e_2 \longrightarrow_b v \\ E' \pm \{x \mapsto v\} \ \vdash \ e \longrightarrow_c v' \end{array}}{E \vdash (e_1\ e_2) \longrightarrow_{1+a+b+c} v'} \qquad \textbf{(app)}$$

$$\frac{E \vdash e_1 \longrightarrow_a \mathit{true} \quad E \vdash e_2 \longrightarrow_b v}{E \vdash (\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) \longrightarrow_{1+a+b} v} \qquad \textbf{(if-true)}$$

$$\frac{E \vdash e_1 \longrightarrow_a \mathit{false} \quad E \vdash e_3 \longrightarrow_b v}{E \vdash (\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) \longrightarrow_{1+a+b} v} \qquad \textbf{(if-false)}$$

$$\frac{}{E \vdash \texttt{nil} \longrightarrow_0 \mathit{nil}} \qquad \textbf{(nil)}$$

$$\frac{E \vdash e_1 \longrightarrow_a v \quad E \vdash e_2 \longrightarrow_b l}{E \vdash (\texttt{cons}\ e_1\ e_2) \longrightarrow_{1+a+b} \langle v, l \rangle} \qquad \textbf{(cons)}$$

$$\frac{E \vdash e \longrightarrow_a \langle v, l \rangle}{E \vdash (\texttt{hd}\ e) \longrightarrow_{1+a} v} \qquad \textbf{(hd)}$$

$$\frac{E \vdash e \longrightarrow_a \langle v, l \rangle}{E \vdash (\texttt{tl}\ e) \longrightarrow_{1+a} l} \qquad \textbf{(tl)}$$

$$\frac{E \vdash e \longrightarrow_a \mathit{nil}}{E \vdash (\texttt{isnull}\ e) \longrightarrow_{1+a} \mathit{true}} \qquad \textbf{(isnull-true)}$$

$$\frac{E \vdash e \longrightarrow_a \langle v, l \rangle}{E \vdash (\texttt{isnull}\ e) \longrightarrow_{1+a} \mathit{false}} \qquad \textbf{(isnull-false)}$$

Figure 3: Standard objects and semantics S with time annotations.

$$\begin{array}{rcl}
b & \in & \textsc{Bool}^{\mathcal{A}} = \{\mathit{true}, \mathit{false}\} \\
L_k & \in & \textsc{List}^{\mathcal{A}} = \{L_0, L_1, \ldots\} \text{ where } L_k \text{ denotes all lists of length} \geq k \\
\left[x, e, E^{\mathcal{A}}\right] & \in & \textsc{Clos}^{\mathcal{A}} = \textsc{Var} \times \textsc{Exp} \times \textsc{Env}^{\mathcal{A}} \\
v & \in & \textsc{DVal}^{\mathcal{A}} = \textsc{Bool}^{\mathcal{A}} + \textsc{List}^{\mathcal{A}} + \textsc{Clos}^{\mathcal{A}} \\
V & \in & \textsc{DValSet}^{\mathcal{A}} = \text{Fin}(\textsc{DVal}^{\mathcal{A}}) + \top^{\mathcal{A}} \\
E^{\mathcal{A}} & \in & \textsc{Env}^{\mathcal{A}} = \textsc{Var} \xrightarrow{fin} \textsc{DValSet}^{\mathcal{A}}
\end{array}$$

Figure 4: Dynamic objects of the abstract semantics $\mathcal{A}$.

time annotations that indicate the number of e-units that an expression's evaluation requires. The evaluation relation $E \vdash e \longrightarrow_i v$ (where $E \in \textsc{Env}$, $e \in \textsc{Exp}$, $v \in \textsc{DVal}$, and $i \in \mathbf{Z}$) indicates that the evaluation of expression e to value v with respect to environment E requires i e-units. For example, the **app** rule states that if the evaluation of e_1 to v_1 requires a e-units, the evaluation of e_2 to v_2 requires b e-units, and the application of v_1 to v_2 requires c e-units, then the evaluation of the application $(e_1\ \ e_2)$ requires $1 + a + b + c$ evaluation units. Similarly, conditional evaluation (**if** rule) counts e-units only in the evaluation of the branch expression selected by the conditional's predicate. Note that the evaluation of λ_v's value terms (*e.g.*, variables and λ-abstractions) requires zero e-units under this relation; a specific implementation would, however, use an e-unit measure and evaluation rules that reflect their concrete costs.

3.1.2 Abstract Semantics $\mathcal{A}$

A non-standard (abstract) semantics $\mathcal{A}$ that abstracts lists as their lengths is used for counting e-units for dynamic granularity estimation. This analysis determines whether an application $(f\ \ l)$ always requires at least i (where $i \geq 0$) e-units of evaluation for a given length of l. The dynamic objects of the abstract semantics are in Figure 4. Every abstract object V denotes a set of values of the standard semantics $\sigma(V)$.[3] A list of length k in the abstract semantics is represented by L_k, the set of all lists with at least k elements.[4] An environment ($\textsc{Env}^{\mathcal{A}}$) maps a program variable either to a concrete finite subset of values or to any such subset (denoted $\top^{\mathcal{A}}$).

The upper bound operation $\sqcup$ on dynamic objects X and Y is defined:

$$X \sqcup Y = \begin{cases} \top^{\mathcal{A}} & \text{if } X = \top^{\mathcal{A}} \text{ or } Y = \top^{\mathcal{A}} \\ \{L_i\} & \text{if } X = \{L_i\} \text{ and } Y = \{L_j\} \text{ and } i \leq j \\ X \cup Y & \text{otherwise} \end{cases}$$

The operator $\sqcup$ is set union, except that $\top^{\mathcal{A}}$ absorbs all other values.

The abstract evaluation relation $E^{\mathcal{A}} \vdash e \xrightarrow{\mathcal{A}}_i V$ (where $E^{\mathcal{A}} \in \textsc{Env}^{\mathcal{A}}$, $e \in \textsc{Exp}$, $V \in \textsc{DValSet}^{\mathcal{A}}$, and $i \in \mathbf{Z}$) evaluates the expression e with respect to (abstract) environment $\mathrm{E}^{\mathcal{A}}$ to a *set* of values V. This relation is defined such that when $e \xrightarrow{\mathcal{A}}_i V$ and $e \longrightarrow_j v$ then $v \in \sigma(V)$ and $i \leq j$. That is, the set of values computed by the abstract relation always contains e's actual value (as produced by $\mathcal{S}$). Furthermore, the e-unit count produced by the abstract semantics is conservative; standard evaluation of e under $\mathcal{S}$ always requires at least i e-units when abstract evaluation of e under $\mathcal{A}$ requires i e-units.

[3] The definition of σ is standard and we omit it here.

[4] Note that L_0 describes all lists and $L_i \supset L_{i+1}$, $i \geq 0$.

$$\frac{}{E^{\mathcal{A}} \vdash e \xrightarrow{\mathcal{A}}_0 \top^{\mathcal{A}}} \qquad (\textbf{any}^{\mathcal{A}})$$

$$\frac{x \mapsto V \in E^{\mathcal{A}}}{E^{\mathcal{A}} \vdash x \xrightarrow{\mathcal{A}}_0 V} \qquad (\textbf{var}^{\mathcal{A}})$$

$$\frac{}{E^{\mathcal{A}} \vdash (\lambda x.e) \xrightarrow{\mathcal{A}}_0 \left\{\left[x, e, E^{\mathcal{A}}\right]\right\}} \qquad (\textbf{abs}^{\mathcal{A}})$$

$$\frac{\begin{array}{rcl} E^{\mathcal{A}} & \vdash & e \xrightarrow{\mathcal{A}}_a \left\{\left[x_1, e_1, E_1^{\mathcal{A}}\right], \ldots, \left[x_n, e_n, E_n^{\mathcal{A}}\right]\right\} \\ E^{\mathcal{A}} & \vdash & e' \xrightarrow{\mathcal{A}}_b V \\ E_i^{\mathcal{A}} \pm \{x_i \mapsto V\} & \vdash & e_i \xrightarrow{\mathcal{A}}_{c_i} V_i, \; 1 \leq i \leq n \end{array}}{E^{\mathcal{A}} \vdash (e \; e') \xrightarrow{\mathcal{A}}_{(1+a+b+\min(c_1,\ldots,c_n))} \bigsqcup_{i=1}^{n} V_i} \qquad (\textbf{app}^{\mathcal{A}})$$

$$\frac{E^{\mathcal{A}} \vdash e \xrightarrow{\mathcal{A}}_a \top^{\mathcal{A}} \quad E^{\mathcal{A}} \vdash e' \xrightarrow{\mathcal{A}}_b V}{E^{\mathcal{A}} \vdash (e \; e') \xrightarrow{\mathcal{A}}_{1+a+b} \top^{\mathcal{A}}} \qquad (\textbf{app-}\top^{\mathcal{A}})$$

$$\frac{E^{\mathcal{A}} \vdash e_1 \xrightarrow{\mathcal{A}}_a \{\mathit{true}\} \quad E^{\mathcal{A}} \vdash e_2 \xrightarrow{\mathcal{A}}_b V}{E^{\mathcal{A}} \vdash (\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3) \xrightarrow{\mathcal{A}}_{1+a+b} V} \qquad (\textbf{if-true}^{\mathcal{A}})$$

$$\frac{E^{\mathcal{A}} \vdash e_1 \xrightarrow{\mathcal{A}}_a \{\mathit{false}\} \quad E^{\mathcal{A}} \vdash e_3 \xrightarrow{\mathcal{A}}_b V}{E^{\mathcal{A}} \vdash (\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3) \xrightarrow{\mathcal{A}}_{1+a+b} V} \qquad (\textbf{if-false}^{\mathcal{A}})$$

$$\frac{E^{\mathcal{A}} \vdash e_1 \xrightarrow{\mathcal{A}}_a V_1 \quad E^{\mathcal{A}} \vdash e_2 \xrightarrow{\mathcal{A}}_b V_2 \quad E^{\mathcal{A}} \vdash e_3 \xrightarrow{\mathcal{A}}_c V_3}{E^{\mathcal{A}} \vdash (\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3) \xrightarrow{\mathcal{A}}_{1+a+\min(b,c)} V_2 \sqcup V_3} \qquad (\textbf{if}^{\mathcal{A}})$$

$$\frac{}{E^{\mathcal{A}} \vdash \texttt{nil} \xrightarrow{\mathcal{A}}_0 \{L_0\}} \qquad (\textbf{nil}^{\mathcal{A}})$$

$$\frac{E^{\mathcal{A}} \vdash e_1 \xrightarrow{\mathcal{A}}_a V \quad E^{\mathcal{A}} \vdash e_2 \xrightarrow{\mathcal{A}}_b \{L_i\}}{E^{\mathcal{A}} \vdash (\texttt{cons } e_1 \; e_2) \xrightarrow{\mathcal{A}}_{1+a+b} \{L_{i+1}\}} \qquad (\textbf{cons}^{\mathcal{A}})$$

$$\frac{E^{\mathcal{A}} \vdash e_1 \xrightarrow{\mathcal{A}}_a V \quad E^{\mathcal{A}} \vdash e_2 \xrightarrow{\mathcal{A}}_b \top^{\mathcal{A}}}{E^{\mathcal{A}} \vdash (\texttt{cons } e_1 \; e_2) \xrightarrow{\mathcal{A}}_{1+a+b} \{L_1\}} \qquad (\textbf{cons-}\top^{\mathcal{A}})$$

$$\frac{E^{\mathcal{A}} \vdash e \xrightarrow{\mathcal{A}}_a V}{E^{\mathcal{A}} \vdash (\texttt{hd } e) \xrightarrow{\mathcal{A}}_{1+a} \top^{\mathcal{A}}} \qquad (\textbf{hd}^{\mathcal{A}})$$

$$\frac{E^{\mathcal{A}} \vdash e \xrightarrow{\mathcal{A}}_a \{L_i\}}{E^{\mathcal{A}} \vdash (\texttt{tl } e) \xrightarrow{\mathcal{A}}_{1+a} \left\{L_{\max(0,i-1)}\right\}} \qquad (\textbf{tl}^{\mathcal{A}})$$

$$\frac{E^{\mathcal{A}} \vdash e \xrightarrow{\mathcal{A}}_a \top^{\mathcal{A}}}{E^{\mathcal{A}} \vdash (\texttt{tl } e) \xrightarrow{\mathcal{A}}_{1+a} \{L_0\}} \qquad (\textbf{tl-}\top^{\mathcal{A}})$$

$$\frac{E^{\mathcal{A}} \vdash e \xrightarrow{\mathcal{A}}_a \{L_0\}}{E^{\mathcal{A}} \vdash (\texttt{isnull } e) \xrightarrow{\mathcal{A}}_{1+a} \{\mathit{true}, \mathit{false}\}} \qquad (\textbf{isnull}^{\mathcal{A}})$$

$$\frac{E^{\mathcal{A}} \vdash e \xrightarrow{\mathcal{A}}_a \{L_i\} \quad i > 0}{E^{\mathcal{A}} \vdash (\texttt{isnull } e) \xrightarrow{\mathcal{A}}_{1+a} \{\mathit{false}\}} \qquad (\textbf{isnull-false}^{\mathcal{A}})$$

Figure 5: Abstract semantics $\mathcal{A}$ with time annotations.

Figure 5 gives the operational rules for the abstract semantics using the $\xrightarrow{\mathcal{A}}_i$ evaluation relation. Beginning with an unevaluated term, the abstract rules are run backwards in a goal-directed fashion towards the axioms. When more than one rule may apply (*e.g.*, **isnull**$^{\mathcal{A}}$ versus **isnull-false**$^{\mathcal{A}}$), the more specific rule is chosen.

Foremost, note that the **any**$^{\mathcal{A}}$ rule can always be applied. Rule **any**$^{\mathcal{A}}$ simply evaluates an expression e to any value and incurs no e-unit cost. Therefore, it is a conservative estimate of values and e-units. Note that abstract evaluation can invoke the **any**$^{\mathcal{A}}$ when the premises of no other rule hold. The rule **any**$^{\mathcal{A}}$ is applied if the depth of the proof exceeds the parallelization cutoff value T (§2).

The **var**$^{\mathcal{A}}$ rule retrieves the mapping of a variable from an environment at zero cost. The **abs**$^{\mathcal{A}}$ rule evaluates a λ-abstraction term to a singleton set containing its closure at zero cost. Again, in practice, costs must be calibrated to a particular machine and implementation.

Abstract evaluation of an application $(e\ e')$ with the **app**$^{\mathcal{A}}$ rule first abstractly evaluates e and e'. When e produces a set F of closures, each $f \in F$ is applied to the value set V that e' produces. The e-unit cost of an application is one e-unit (for the application proper), the e-units required for (abstractly) evaluating e and e', and the minimum of the e-unit costs incurred in applying each $f \in F$ to V. This gives a conservative e-unit count because the cost of the least expensive function reaching the application is used. The set of values produced by **app**$^{\mathcal{A}}$ is the union of the value sets produced by the applications of the closures F. The **app**-$\top^{\mathcal{A}}$ rule handles the case where F is not known.

The conditional rules (**if-true**$^{\mathcal{A}}$, **if-false**$^{\mathcal{A}}$, **if**$^{\mathcal{A}}$) conservatively approximate a conditional's behavior. If the predicate abstractly evaluates to a singleton set containing either *true* or *false*, the respective conditional branch is abstractly evaluated. However, when the predicate's abstract value set is not precisely known (*e.g.*, when it contains both *true* and *false*), both conditional branches are abstractly evaluated and the minimum e-unit cost of these evaluations is incorporated into the conditional's cost—the set of values produced by the conditional is the union of the value sets produced by both conditional branches.

The rules for list objects and the primitive list functions operate as follows. The **nil**$^{\mathcal{A}}$ rule evaluates the syntactic constant `nil` to the abstract value that contains the set of all lists ($\{L_0\}$). Abstract evaluation of the constant `nil` incurs no e-unit cost.

A list's size (length) increases when an element is *consed* onto it. List creation with the special cons form (**cons**$^{\mathcal{A}}$ rule)—when the tail of the new list is in the set L_i; *i.e.*, it is a list of at least length i)—produces the set of lists of at least length $i+1$, L_{i+1}. The abstract e-unit cost for this operation is one plus the cost of evaluating the arguments to **cons**. The **cons**-$\top^{\mathcal{A}}$ rule handles the case where all information about the list being *consed* onto has been lost.

Selecting the head (**hd**$^{\mathcal{A}}$ rule) of any object returns any value ($\top^{\mathcal{A}}$) since a list's contents (its elements) are not maintained in the abstract semantics. Selecting the tail (**tl**$^{\mathcal{A}}$ rule) of a list of at least length i returns L_{i-1}, the set of lists of at least length $i-1$, since the list returned by the tail selector is always one less than the length of its argument list. The **tl**-$\top^{\mathcal{A}}$ rule handles application of `tl` to an unknown list.

Testing for the empty list with `isnull` produces the set $\{false\}$ when `isnull`'s argument is a list of at least length ≥ 1 (**isnull-false**$^{\mathcal{A}}$ rule). Otherwise, this test conservatively returns $\{true, false\}$ under abstract evaluation (**isnull**$^{\mathcal{A}}$ rule).

3.1.3 Termination

Counting e-units in the abstract semantics $\mathcal{A}$ proceeds conservatively along both arms of a conditional whose abstract predicate value is imprecise (*i.e.*, neither $\{true\}$ nor $\{false\}$). This ensures that the cost of an expression is conservatively approximated as the cost of its least-costly execution path. Doing so, however, introduces the possibility of non-termination under abstract evaluation since abstract evaluation can now attempt to evaluate a term that diverges under the standard semantics.

This termination problem is solved by bounding the number of abstract evaluation steps. Evaluation of an execution path under $\mathcal{A}$ terminates (along that path) when the accumulated e-units exceed the overhead threshold T (§2). In other words, when viewed as a deductive proof, the proof tree of an expression's abstract evaluation never exceeds a depth of T unit-cost deductions; *i.e.*, the **any**$^{\mathcal{A}}$ rule is applied upon reaching this bound. Halting abstract evaluation in this manner avoids the non-termination issue since we only evaluate for a bounded T e-units along any execution path and return the cost of the least-cost path.

3.1.4 Program Restructuring

A compiler can use dynamic granularity estimation to restructure the program as follows. The compiler wraps a conditional around every application expression, `(f l)`, that applies function `f` to a list `l`. The conditional's branches respectively contain code for the sequential and parallel evaluation of the application expression (see, for example, Figure 2). The predicate of the compiler-supplied conditional examines the length of `l` (available at run time) and compares it to a compiler-deduced cutoff value (described below). When `l`'s length is at least equal to this cutoff, the conditional selects parallel evaluation for `(f l)`.

The compiler deduces the cutoff value using abstract evaluation in the following manner. Suppose that **dge**'s dynamic component (§3.2) precisely maintains the lengths of all lists of length $< n$, and that all lists with lengths $\geq n$ are approximated as such. The compiler abstractly evaluates `(f` $\{L_i\}$`)` for $0 \leq i < n$. When `(f` $\{L_i\}$`)` $\xrightarrow{\mathcal{A}}_x V$, it notes the least i such that the cost x of this application is always greater than the overhead threshold T. This least i, if it exists, represents a length cutoff for `l` at which the creation of a parallel thread for `(f l)` is always beneficial. The value of this least i is the cutoff value in the conditional guarding the application.

In general, the compiler can use the abstract-evaluation semantics to determine a cost threshold for *any* expression e, not just for the application of functions to lists. To do so, it must first identify all lists in e; it then abstractly evaluates e for all list-length combinations and notes the lengths at which parallel evaluation of e is viable. This list-length information is then used to construct a predicate to select sequential or parallel evaluation for e.

3.2 Dynamic Component

At run-time, **dge**'s dynamic component maintains an approximation to the length of a list l along with l's physical representation. An implementation that represents lists with *cons cells* in a heap is assumed. A fixed field of b bits encodes length information. This allows lists of length $< 2^b - 1$ to be exactly represented. Approximate lists have length ∞; that is, an approximate list is of length $\geq 2^b - 1$. When a new list is formed with the list constructor, as in $l \equiv (\text{cons } x\ l')$, the length field on l is set to $\overline{l'} + 1$ if $\overline{l'}$ is not ∞. Otherwise, it is set to ∞.

An implementation of the dynamic component can store the b bits of length information either in a cons cell, or in the pointers to a cons cell. Storing the approximation within the cell requires an additional memory access when forming a new cell since the length field pointed to by the new cell's tail pointer must be fetched. If the cons-cell representation does not contain b unused bits, additional storage must also be allocated in the cell under the first scheme. The second approach requires the pointer representation to contain b unused bits, but avoids an additional memory fetch since construction of a new cons cell always requires the pointer to the list that becomes the new cell's tail field. The first approach is significantly simpler to implement because it only requires modification to the portion of the compiler that generates the code for cons-cell creation (§5). The second approach requires modifications to the implementation's run-time system (*e.g.*, the garbage collector), the generation of special pointer dereferencing code, and (potentially) a revision of the memory layout.

The final concern in the design of the dynamic component is, how many bits, b, to allocate for the length field. A value for b is best selected by consulting the empirical results of applying **dge**'s static analysis (§3.1) to actual programs because, for a typical application $(f\ l)$, where $|(f\ l)|$ depends on the length of l, it is likely that a threshold value for $\bar{l}$ exists at which parallel evaluation of $(f\ l)$ is fruitful. The number of bits b should be large enough to delineate this threshold for most cases.

4 Implementation

The dynamic component of dynamic granularity estimation has been implemented in the Standard ML of New Jersey 0.73 optimizing compiler [2]. The MP queue-based multiprocessing platform [17, 3] provides thread creation, synchronization, and management primitives. The *sml2c* code generator [22] outputs C code for execution on a 20-processor shared-memory Sequent Symmetry.

The compiler and run-time system were modified to incorporate one machine word (32 bits) of length information into the standard (three-word) representation of every cons cell (*cf.* §3.2). The compiler's front end was modified to distinguish cons cells from all other types of dynamic objects. This modification identifies cons cells as such for the compiler's back end. The code generator was modified to produce code that computes list lengths upon cons-cell formation. Since a list's length is represented by a full machine word, code for approximating list lengths is unnecessary and is not generated. We introduced high-level functions to provide access to a list's length information. Such integer lengths can be manipulated as ML values and compared against the overhead-threshold value (determined empirically, §2). Low-level primitives, *i.e.* abstract machine instructions, would provide even better performance.

The static component for **dge** has not been implemented—abstract evaluation was performed manually.

5 Results

Figure 6 gives the results of dynamic granularity estimation applied to a quicksort (qs, Figure 2) sorting a list of 10000 random integers. The recursive applications of qs for sorting sublists were performed in parallel on 8 processors.[5] The graph plots list-length cutoffs versus execution time. Here, we examine the effect of varying qs's list-length cutoff value on the program's execution time. Parameters of a specific language implementation and machine architecture would enable **dge**'s static component to automatically select a concrete cutoff.

Execution, garbage collection, and total times are given for qs with and without **dge**. The graph's top two curves are the total time required with dynamic granularity estimation (dge) and with standard parallel evaluation (std) respectively. The x-axis is the cutoff values at which threads are retained for sequential evaluation. For the **dge** times, a length cutoff i indicates that

[5]The standard parallel execution time of the graph represents a speedup of 3.8 (on 8 processors) over standard sequential execution.

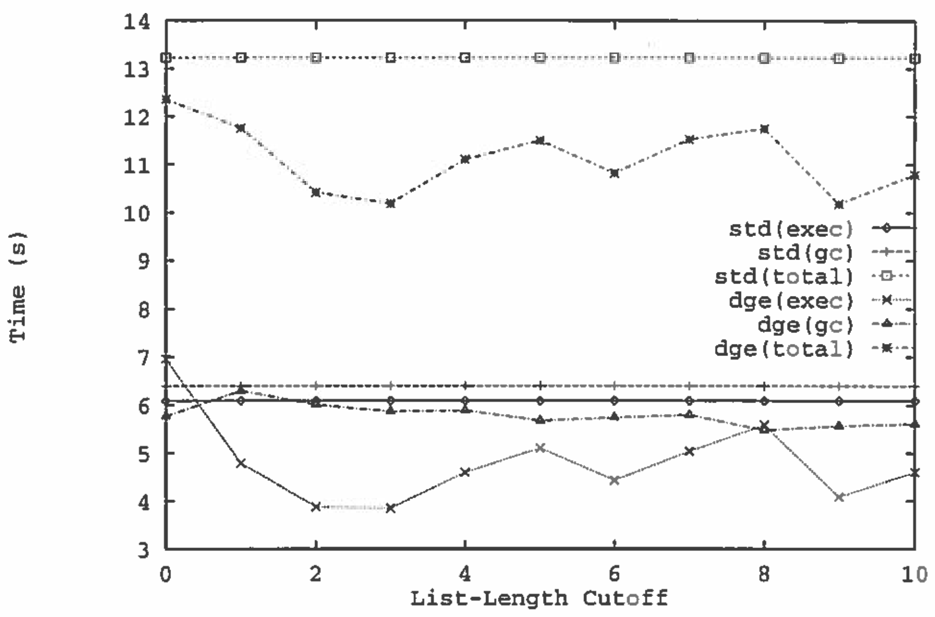


Figure 6: Effect of varying the list-length cutoff threshold in parallel evaluation with 8 processors of quicksort (Figure 2).

the arguments to append in qs evaluate in parallel only when the lengths of the sublists bound to l and g both equal or exceed i. The (std) times are for an ML implementation without the modifications and associated overhead for maintaining list-lengths at run time. The graph's lower curves break the total time into execution (exec) and garbage collection (gc) times. Time spent in the operating system are included in the total times.

Dynamic granularity estimation improves qs's performance at all cutoff values i, $0 \leq i \leq 10$. If thread creation is throttled when sublists are of length < 3, dge reduces the total time to execute the program by $\approx 23\%$. Figure 6 also reveals that garbage collection times slightly decrease as the cutoff length increases—fewer threads require fewer memory resources.

6 Related Work

We do not know of previous approaches that estimate the amount of computation in an expression by examining dynamic information. Aside from simple heuristics [8], work related to (static) granularity estimation falls into one of two categories: load-balancing strategies that continually monitor the number of active threads in the machine to determine when it saturates, and systems that statically derive an algorithm's time complexity, if possible.

In Halstead's Multilisp [9, 10], the program ceases to create new parallel threads when the machine saturates with threads. When this occurs, processors evaluate the available threads to completion. Idle processors steal threads from busy processors in this load-based inlining scheme. Load-based inlining, in the presence of Multilisp's futures, poses deadlock problems, but these can be avoided by Mohr *et al.*'s lazy task creation technique [16, 15]. Lazy task creation efficiently extracts computation from inlined threads when no runnable threads exist. Although lazy task creation increases the granularity of programs by coalescing threads, unlike dge, it does not prevent the production of fine-grain threads that are detrimental to the program's quick evaluation. WorkCrews [24] is a thread management package that performs lazy task creation, but requires programmer knowledge of the mechanism. Qlisp [6] provides primitives for performing load-based thread creation as well as automatic load-based inlining [18].

Dynamic granularity estimation is a *load-insensitive* technique that only creates parallel threads that are guaranteed to meet or exceed some granularity criterion. Therefore, dge is orthogonal—and complements—existing load-based inlining and task creation methods. Harrison's parallel Lisp system, PARCEL [11], employs a non-standard list representation that dynamically maintains information about a list's length. PARCEL uses length information to implement lists contiguously in memory, but not for making parallelization or load-balancing decisions. METRIC [25] statically transforms Lisp programs into a set of mutually recursive equations and then seeks their solution to yield the program's complexity. Le Métayer's ACE complexity evaluator [14] matches list-based functional programs against a predefined library of function definitions to map programs to their time complexities. Sands extended this approach to higher-order lazy languages [21]. Dornic *et al.* [5] describe a practical *time system* that statically infers a function's complexity from its local definition; *i.e.*, their analysis does not require interprocedural information. Their time system, however, is conservative since it approximates recursive functions as always being expensive to evaluate. In contrast to dynamic granularity estimation, static time-complexity analyses cannot accurately predict an expression's cost when dynamic data sizes are not known at compile time.

We have previously used run-time information to dynamically discover parallelism in imperative higher-order programs that build and modify dynamic data structures [13, 12].

Acknowledgement

This work was supported in part by the National Science Foundation under grant CCR-9101035 and by the Wisconsin Alumni Research Foundation. L. Huelsbergen was supported by an ARPA fellowship in parallel processing.

References

[1] S. Abramsky and C. L. Hankin, editors. *Abstract Interpretation of Declarative Languages.* Ellis Horwood Ltd., Chichester, West Sussex, England, 1987.

[2] A. W. Appel and D. B. MacQueen. A Standard ML compiler. *Functional Programming Languages and Computer Architecture*, 274:301–324, 1987.

[3] E. C. Cooper and J. G. Morrisett. Adding threads to Standard ML. Technical Report CMU-CS-90-186, School of Computer Science, Carnegie Mellon University, December 1990.

[4] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Symposium on Principles of Programming Languages*, pages 238–252. Association for Computing Machinery, 1977.

[5] V. Dornic, P. Jouvelot, and D. K. Gifford. Polymorphic time systems for estimating program complexity. *ACM Letters on Programming Languages and Systems*, 1(1):33–45, March 1992.

[6] R. P. Gabriel and J. McCarthy. Queue-based multi-processing Lisp. In *Lisp and Functional Programming*, pages 25–44. Association for Computing Machinery, August 1984.

[7] R. Goldman and R. P. Gabriel. Qlisp: Experience and new directions. In *Proceedings of ACM/SIGPLAN PPEALS 1988 (Parallel Programming: Experience with Applications, Languages and Systems)*, pages 111–123, July 1988.

[8] S. L. Gray. Using futures to exploit parallelism in Lisp. Master's thesis, MIT, February 1986.

[9] R. H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

[10] R. H. Halstead, Jr. An assessment of Multilisp: Lessons from experience. *International Journal of Parallel Programming*, 15(6):459–501, 1986.

[11] W. L. Harrison and D. A. Padua. PARCEL: Project for the automatic restructuring and concurrent evaluation of lisp. In *International Conference on Supercomputing*, pages 527–538, July 1988.

[12] L. Huelsbergen. *Dynamic Language Parallelization*. PhD thesis, University of Wisconsin–Madison, August 1993.

[13] L. Huelsbergen and J. R. Larus. Dynamic program parallelization. In *Lisp and Functional Programming*, pages 311–323. Association for Computing Machinery, June 1992.

[14] D. Le Métayer. ACE: An automatic complexity evaluator. *ACM Transactions on Programming Languages and Systems*, 10(2):248–266, April 1988.

[15] E. Mohr. *Dynamic Partitioning of Parallel Lisp Programs.* PhD thesis, Yale University, August 1991.

[16] E. Mohr, D. Kranz, and R. H. Halstead, Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. In *Lisp and Functional Programming*, pages 185–197. Association for Computing Machinery, June 1990.

[17] J. G. Morrisett and A. Tolmach. Procs and locks: A portable multiprocessing platform for Standard ML of New Jersey. In *Principles and Practice of Parallel Programming*, pages 198–207. Association for Computing Machinery, May 1993.

[18] J. D. Pehoushek and J. S. Weening. Low-cost process creation and dynamic partioning in Qlisp. In *US/Japan Workshop on Parallel Lisp*, pages 183–199. Lecture Notes in Computer Science, June 1989.

[19] G. D. Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[20] J. C. Reynolds. GEDANKEN—a simple typeless language based on the principle of completeness and the reference concept. *Communications of the ACM*, 13(5):308–319, 1970.

[21] D. Sands. Complexity analysis for a lazy higher-order language. In *ESOP*, pages 361–376. Lecture Notes in Computer Science, May 1990.

[22] D. Tarditi, A. Acharya, and P. Lee. No assembly required: Compiling Standard ML to C. Technical Report CMU-CS-90-187, School of Computer Science, Carnegie Mellon University, November 1990.

[23] M. Tofte. *Operational Semantics and Polymorphic Type Inference*. PhD thesis, University of Edinburgh, Department of Computer Science, May 1988.

[24] M. T. Vandevoorde and E. S. Roberts. WorkCrews: An abstraction for controlling parallelism. *International Journal of Parallel Programming*, 17(4):347–366, 1988.

[25] B. Wegbreit. Mechanical program analysis. *Communications of the ACM*, 18(9):528–539, September 1975.

Generators, sharing and freeness: the theory and practice of deriving data-dependencies

Bob Kemp and Andy King

Department of Electronics and Computer Science,
The University of Southampton, S09 5NH, UK.
e-mail: {rsk,amk}@ecs.soton.ac.uk

Abstract

Generators are a new approach for reducing the time complexity of the abstract interpretation of concurrent logic programs. Instead of naively abstracting all reduction sequences, the generator approach produces for each branch of the proof tree a set of the branches which must be executed before it. This, in addition to tracing sharing behaviour, provides valuable data-dependency information for program optimisation. To be precise, the generator semantics is an abstract interpretation framework which is parameterised by a domain and an abstract equation solver. The paper briefly reviews the generator approach, synthesises an efficient freeness analysis, and finally presents some initial experiments applying the freeness analysis within the generator framework.

1 Introduction

Concurrent logic programming brings a new dimension of expressiveness to logic programming enabling a host of useful protocols and paradigms to be modelled. This flexibility does not come without cost, however, since it is the control strategy of Prolog that brings an efficient stack based implementation. Concurrent logic programs, on the other hand, incur overheads from scheduling, argument copying and an increased memory turnover. To reduce these overheads a number of compilation techniques have been proposed which (essentially) take as input the data-dependencies between the atoms of a clause and produce as output an optimised program. Thread based compilation schemes [10, 14, 13], for instance, automatically introduce control-flow into data-flow logic programs by applying conservative data-dependency information. Threading boils down to deducing at compile-time a partial schedule of processes, or equivalently the body atoms of a clause, which is consistent with the program behaviour. To avoid compromising program termination, an ordering of the atoms is determined which does not contradict any data-dependence of the program. In general the processes cannot be totally ordered and thus the analysis leads to a division into threads of totally ordered processes. In this way the work required of the run-time scheduler is reduced to ordering threads. In addition to avoiding the creation of unnecessary suspensions, the ordered nature of a thread permits useful Prolog style optimisations to be introduced into the program [6, 9]. Threading variants, such as serialisation analysis [11], (which applies a granularity control in conjunction with threading), in addition to requiring data-dependency information, need granularity information too! In short, data-dependency information is vital for underpinning these techniques.

Other techniques, such as local reuse [5], make less overt use of data-dependency information. Local reuse can efficiently recycle memory without the use of free-lists and garbage collection for terms which are only referenced by one producer process and one consumer process. The single producer and single consumer constraint, in effect, specifies a particular form of data-dependency which can be easily recognised by analysis [19] and yet can aid garbage collection. More generally, reformulating the problem of reuse in terms of recognising the last consumer of a variable, may well yield more precise analysis. Reasoning about the last consumer of a variable, however, requires data-dependency information.

Analysing concurrent logic programs, however, can be expensive due to the interleaving of sub-goals. The generator semantics avoids an explosion of cases by keeping independent sub-computations separate until synchronisation is required. This, in effect, factors out the interleaving and therefore provides a tractable approach to reasoning about concurrency. Synchronisation induces data-dependencies which can be collected for use in threading and other optimisations.

The layout of the paper is organised as follows. Section 2 outlines the generator semantics. The generator semantics, in effect, defines an abstract interpretation framework. The framework is parameterised by a domain and an abstract equation solver so that fleshing out these two components finally defines a complete abstract interpreter. Section 3 introduces an efficient freeness analysis which is used in the current prototype implementation. Section 4 sketches the implementation and briefly presents some initial results.

2 Generators

The execution of a flat concurrent logic language program may be viewed as the building of a sequence of partial proof trees. To begin with there is just a root representing the initial goal. This is extended with some program clause and the body atoms of the clause replace their parent in the frontier of the tree. This process is repeated until there are no leaves in the frontier that can be further extended.

Because of interleaving, different atoms in the frontier can potentially be extended at any stage. On the other hand, some atoms may be insufficiently instantiated for this to happen, either because the sub-goal is less instantiated than every clause head, or because the guards are not satisfied. Extension by some clauses, such as =/2, can bind variables and allow other leaves of the tree to be extended. In this way, producers and consumers synchronise.

This model gives an intuition of how execution progresses and suggests an alternative way of viewing it. The computation is composed of many, largely-independent, sub-computations. The growth of each branch charts the progress of a sub-computation. Synchronisation between these sub-computations define the data-dependencies. In order to reason about the data-dependencies, it is convenient to keep independent sub-computations separate until synchronisation is required. This, in effect, means keeping independent branches of the proof trees separate. (Partial proof trees can be reconstructed by unifying their branches together.) These independent sub-computations are called generator trees (a branch is a pruned tree).

Computation can be regarded as a sequence of partial proof trees. However, to reason about data-dependencies, it is more useful to trace the corresponding sequence of sub-computations. Put another way, an inductive definition of the sequence of generator trees is required. To define a generator the notion of the tip of a generator is needed. Intuitively, the tip is the last sub-tree of a generator that was extended, that is, the clause which it was extended by.

A generator tree is an independent sub-computation. Put another way, a generator cannot include branches which are not required to for the sequence of extensions required to produce its tip. If the generator included extra branches it could be divided into two or more independent sub-computation. This, in effect, induces a minimality constraint which must be satisfied when extending generators. To be exact, a generator is the minimal tree required to produce the extensions leading to its tip. This is the essence of the generator idea. In fact, because of minimality, the generators for an initial goal can be unified to produce an arbitrary state in the execution of that goal.

Because of minimality, generators can only be extended at their tip. For each leaf of the tip, extending a generator at that leaf by some clause can potentially produce a new generator with the body goals of the clause as its tip. In some cases, an extension may not be possible due to the leaf atom being insufficiently bound. In this case, one or more other generators must be unified with the present generator to provide the necessary bindings. Only the minimal number of other generators must be used else minimality will be violated.

2.1 Interleaving Semantics

Let VARS be the set of all variables, TERMS the set of all terms and ATOMS be the set of all atoms formed from some first-order language. This hierarchy is extended to expressions, EXPR, with the introduction of nested tuples of atoms (denoted by angle brackets). Expressions are used to represent clauses and partial proof trees. The variance relation on EXPR, denoted $\approx$ ($e_1 \approx e_2$ if e_1 and e_2 are related by a renaming of variables) is an equivalence relation. Let $[e]$ denote the equivalence class of $e \in$ EXPR and $\text{EXPR}_{\approx}$ be the quotient set $\text{EXPR}/\approx$. Reynolds [17] showed that term algebras like $\text{EXPR}_{\approx}$, with an added top element form a complete lattice $(\text{EXPR}_{\approx} \cup \{\top\}, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$. VARS is the least element of this lattice; the join operation, $\sqcup$, corresponds to unification and the ordering $\sqsubseteq$ to the "is less instantiated than". Thus $[f(a,y)] \sqcup [f(x,g(x))] = [f(a,g(a))]$ while $[f(a,y)] \sqcup [f(b,g(x))] = \top$, where letters at the end of the alphabet denote variables.

Partial proof trees are represented by $\text{TREES}_{\approx}$, a sub-lattice of $(\text{EXPR}_{\approx} \cup \{\top\}, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$ defined by:

$$\begin{aligned}
\text{Labels} &= \text{VARS} \cup \text{ATOMS} \cup \{\langle\rangle\} \\
T_0 &= \{\langle h, \_ \rangle \mid h \in \text{Labels}\} \\
T_k &= T_{k-1} \cup \{\langle h, \langle t_1, \ldots, t_n \rangle\rangle \mid h \in \text{Labels}, n \geq 0, t_1, \ldots, t_n \in T_{k-1}\} \\
\text{TREES}_{\approx} &= \{\top\} \cup \{[t] \mid j \geq 0, t \in T_j\}
\end{aligned}$$

Because head matching and guard execution complete before the reduction of the body goals begin, guards do not need to be explicitly represented in the proof tree. A program clause $h \leftarrow g \mid b_1, \ldots, b_n$, thought of as a partial proof tree, is represented by the equivalence class $[\langle h, \langle\langle b_1, \_\rangle, \ldots, \langle b_n, \_\rangle\rangle\rangle] \in \textsc{Trees}_{\approx}$. Anonymous variables are represented by '_', as in Prolog. A unit clause $h \leftarrow$ is represented by the class $[\langle h, \langle\rangle\rangle]$ and is depicted as a *closed leaf.* The set of program clauses W will then be a subset of $T_1/\approx$, with goals represented by trees with a root labelled by '$\langle\rangle$', such as $[\langle\langle\rangle, \langle\langle a, \_\rangle\rangle\rangle]$ for $\leftarrow a$. A generic goal can be represented by the expression $[\langle\langle\rangle, \_\rangle]$. The set of program and predefined clauses is $W' = W \cup \{[\langle x = x, \langle\rangle\rangle]\}$.

Extremities of trees of the form $\langle t, \_\rangle$ are called *open leaves* because a partial proof tree can be extended at these positions; those of the form $\langle t, \langle\rangle\rangle$ are called *closed leaves.* The tuple of labels for a tree's open leaves (taken in order, from left to right) is called the *frontier* of the tree. The definite clause represented by a tree $[t]$ is recovered as "[rootlabel$(t) \leftarrow$ frontier(t)]".

The frontier leaf selected for extension can be specified by a path from the root of a partial proof tree to that leaf.

Definition 1 *Let* $\textsc{Paths}$ *be the set of all strings of positive integers, including the empty string,* λ. *A* path *is used to describe the journey from the root to some subtree. The partial function* $! : \textsc{Expr}_{\approx} \times \textsc{Paths} \rightarrow \textsc{Expr}_{\approx}$ *(pronounced pling) is recursively defined by*

$$[\langle t_1, \ldots, t_j, \ldots, t_n\rangle]\,!\,(j.p) = [t_j]\,!\,p \qquad [t]\,!\,\lambda = [t]$$

where '.' is the path 'cons' operation.

An assumption, which is not essential but simplifies proofs, is that the tuple *representations* of different program clauses are not unifiable.

Definition 2 *Let S be a subset of* $\textsc{Trees}_{\approx}$. *Then* $[t], [t'] \in S$ *are said to be* disjoint *if* $[t] \neq [t'] \Rightarrow [t] \sqcup [t'] = \top$

The clause trees in W' are henceforth assumed to be disjoint. If this is not the case, clauses can be indexed to achieve the same effect.

A immediate extension step $[t] \xrightarrow{p} [t']$, for $[t], [t'] \in \textsc{Trees}_{\approx}$, is one that is allowed by the computation rule. A formal definition in this notation can be found in [8].

Definition 3 *let* $[s]$, $[t]$ *and* $[t']$ *be partial proof trees. Let* $p \in \textsc{Paths}$. *Let* $W' \cup \{\langle\langle\rangle, \_\rangle\}$ *be a set of disjoint clause trees. Then let* $\longrightarrow \subseteq \textsc{Trees}_{\approx} \times \textsc{Paths} \times \textsc{Trees}_{\approx}$ *be a relation such that* $[t] \xrightarrow{p} [t']$ *iff* $[t']$ *is an allowed extension of* $[t]$ *at* $[t]\,!\,p$ *by some program clause and if* $[t]$ *and* $[t']$ *are extensions of* $[s]$ *(i.e.* $[s] \longrightarrow^* [t]$ *and* $[s] \longrightarrow^* [t']$*) then* $[s] \longrightarrow^* [t] \sqcup [t']$ *where* $\longrightarrow^*$ *be the least reflexive, transitive relation satisfying* $[t] \longrightarrow^* [t']$ *if* $[t] \xrightarrow{p} [t']$.

The set $\textsc{Trees}_{\approx}$ ordered by $\longrightarrow^*$ forms a partial order. By extending the ordering to make $\perp$ and $\top$ the least and greatest elements, a complete lattice is produced. The generator semantics of the next subsection combines trees with the join operation of this lattice and crucially depends on the partial ordering. If the clauses in W are disjoint, the join in this lattice corresponds to $\sqcup$, although the meet is in general not the same.

Definition 4 *Let $W' \cup \{[\langle\langle\rangle, \_\rangle]\}$ be a set of disjoint clauses. Define* $\trianglelefteq \subseteq \text{TREES}_{\approx} \times \text{TREES}_{\approx}$ *by* $\forall [t], [t'] \in \text{TREES}_{\approx}$

$$[t] \trianglelefteq [t'] \quad \textit{if} \quad [t] \longrightarrow^* [t']$$
$$[t] \trianglelefteq \top$$
$$\bot \trianglelefteq [t']$$

Theorem 1 *Let $S \subseteq \text{TREES}_{\approx}$ and $W' \cup \{[\langle\langle\rangle, \_\rangle]\}$ be a set of disjoint clause trees. If $\bigsqcup S \neq \top$, then $\bigsqcup S$ is the least ($\sqsubseteq$) common extension of all $[s] \in S$.*

Theorem 2 *If the clause trees in $W' \cup \{[\langle\langle\rangle, \_\rangle]\}$ are disjoint then $(\text{TREES}_{\approx}, \trianglelefteq)$ forms a complete lattice.*

Without any loss of generality, program clauses can be assumed disjoint and so $(\text{TREES}_{\approx} \cup \{\top, \bot\}, \trianglelefteq)$ forms a complete lattice. As such $glb_{\_\_}.\{[t], [t']\}$ will usually be written as $[t] \triangle [t']$ and $lub_{\_\_}.\{[t], [t']\}$ as $[t] \triangledown [t']$. Note that in general $[t] \sqcap [t']$ is not the same as $[t] \triangle [t']$ because $[t]$ and $[t']$ may both have common extensions that would be enabled by different extensions in each. Thus the common extension would be in $[t] \sqcap [t']$, but neither of the two extensions providing the instantiation to enable that common extension would be there.

2.2 Generator Semantics – $\mathcal{O}_g$

Definition 5 *Let W be a set of clauses, $[G]$ the tuple representation of some goal, $p \in \text{PATHS}$ and $[t] \in \text{TREES}_{\approx}$. Then $\langle [t], p \rangle$ is a* generator *based on $[G]$ if $[t]$ is a minimal ($\trianglelefteq$) tree such that $[G] \trianglelefteq [t]$ and that $[t]\,!\,p$ is an instance of a clause in $W \cup \{[G]\}$. $[t]\,!\,p$ is called the* tip *of the generator.*

Generators are denoted by $\langle [s], p \rangle$ where $[s]$ is the tree and $[s]\,!\,p$ is its tip. Intuitively, a *generator* is produced by always extending children of the most recently extended branch (depth-first), resulting in tall, thin partial proof trees. They are called generators because all partial proof trees can be recovered (generated) from joins of such tall, thin partial proof trees:

$$\beta : \wp(\text{TREES}_{\approx} \times \text{PATHS}) \rightarrow \wp(\text{TREES}_{\approx})$$
$$\beta(S) = \{[s] \mid S' \subseteq S, [s] = \triangledown\{[t] \mid \langle [t], p \rangle \in S'\}, [s] \neq \top\}$$

In words, β takes a set S of generators to the set of joins of arbitrary subsets of it, i.e. the set of partial proof trees generated by S.

Based on the tree extension relation, '$\longrightarrow$', a new function $\mathcal{O}_g$ can be stated that produces all immediate extensions of a set of generators at their tips.

$$\mathcal{O}_g : \wp(\text{TREES}_{\approx} \times \text{PATHS}) \rightarrow \wp(\text{TREES}_{\approx} \times \text{PATHS})$$
$$\mathcal{O}_g(S) = S \ \cup \ \bigcup_{\langle [t], p \rangle \in S} \min_{\trianglelefteq} \left\{ \langle [r], p.q' \rangle \;\middle|\; \begin{array}{l} [t'] \in \beta(S) \\ \wedge\ ([t] \sqcup [t']) \xrightarrow{p \cdot q'} [r] \end{array} \right\}$$

where $\min_{\trianglelefteq}$ selects the elements with minimal first arguments. In other words, the path is ignored in minimisation. $\mathcal{O}_g(S)$ is the set of trees S plus any immediate extensions of their tips permitted by the computation rule. By design, if $\langle [s], p\rangle$ is a generator and $[s]\,!\,p$ is an open leaf, extending it in this way produces another generator. A consequence of the semantics is data-dependency information from the implicit partial ordering, $\trianglelefteq$, on goal reduction.

Because $\mathcal{O}_g$ constructs trees using ∇ and $\min_{\trianglelefteq}$, the trees are minimal in the following sense.

Theorem 3 *Let* $[G] \in \text{TREES}_{\approx}$ *and* $\langle [t], p\rangle \in \mathcal{O}_g^{(n)}(\{\langle [G], \lambda\rangle\})$. *Then* $[t]$ *is a generator with tip* $[t]\,!\,p$.

From the definition of $\longrightarrow^*$ and a simple inductive argument, every generator in $\mathcal{O}_g(S)$ is an extension of a tree in S. Likewise, due to the minimal number of branches in each generator, any legal extension of an initial goal can be produced as the least common instance of some subset of $\beta(\mathcal{O}_g^{(n)}(S))$ for some n.

Lemma 1 *Let* $[G] \in \text{TREES}_{\approx}$ *represent a goal then*

$$\begin{aligned} \forall n \geq 0 \,.\, [G'] \in \beta(\mathcal{O}_g^{(n)}(\{\langle [G], \lambda\rangle\})) \;&\Rightarrow\; [G] \longrightarrow^* [G'] \\ [G] \longrightarrow^* [G'] \;&\Rightarrow\; \exists n \geq 0 \,.\, [G'] \in \beta(\mathcal{O}_g^{(n)}(\{\langle [G], \lambda\rangle\})) \end{aligned}$$

Theorem 4 *For all* $[t] \in \text{TREES}_{\approx}$, $\beta(\bigcup_{n\geq 0} \mathcal{O}_g^{(n)}(\{\langle [t], \lambda\rangle\}))$ *is the set of all extensions of* $[t]$.

3 Freeness

The generator framework is parameterised by a domain and an abstract equation solver so that fleshing out these two components finally defines the complete abstract interpreter. The domain and abstract equation solver invariably correspond to a sharing or a freeness analysis. Sharing (or aliasing) analysis conventionally infers which program variables are definitely grounded and which variables can never be bound to terms containing a common variable [20, 4, 2, 7]. Freeness analysis, in addition, traces a structural or type property of bindings and substitutions. In short, a freeness analysis infers which variables are free, that is, which variables are definitely not bound to non-variable term [16, 3, 18]. Freeness analysis is particularly important in the generator approach because:

1. Accurate sharing information is required to limit the number of generators which may produce a binding. Put another way, without accurate sharing, the efficiency of the generator approach can degrade badly. Sharing can be accurately traced by tracking definite groundness [7, 16] and definite freeness [15]. Sharing abstractions usually capture groundness and are often good at groundness propagation [7, 16], but without exploiting freeness (or linearity [2]), usually have to assume that aliasing is transitive. The significance of freeness is that the unification of free variables only yields restricted forms of aliasing. Thus, if variables can be inferred to be free, worst case aliasing need not be assumed in an analysis.

2. Without sequentiality to induce a notion of restriction and extension [1], the variables under consideration grow from those of a clause to those of a generator. This, in effect, means that abstract substitutions can grow large in size, and therefore expensive operations on abstract substitutions, like the closure under union operation of [7, 16], should be avoided whenever possible. One way to apply closure operations more conservatively is to track freeness. Freeness simplifies the manipulation of abstract substitutions and thus contributes to faster analysis.

3. Freeness is important for reasoning about matching and synchronisation. If an argument of a sub-goal can be inferred to be free, and all (renamed) clauses of the corresponding predicate at that argument position are non-variable terms, the sub-goal may suspend. In terms of the generators, if a generator leaves a variable free which is required to be non-free to reduce the clause, then that generator must be combined with other generators to allow a reduction. This, in turn, can improve precision because in unifying the generators, variables may be grounded and thus extraneous generators can be ruled out when the resulting generator is itself extended. In short, accurate freeness information improves precision.

As part of an initial feasibility study, an accurate but efficient freeness analysis has been synthesised. The rationale behind the construction of the analysis was that if the generator semantics was not found to be tractable with a particularly efficient freeness component, then the semantics was unlikely to be practical for more complicated abstract unification algorithms. This particular freeness analysis adopts a $Share \times Free$ style domain in the spirit of [15] but instead of using proposed $aunify$ function, applies a particularly simple abstract equation solver which avoids the calculation of a complicated fixed-point operations whenever possible. In short, the rationale behind the reformulation is speed.

The exposition is structured as follows. In section 3.1, the focus is on abstracting data, and an abstraction of substitutions for freeness is formalised. In section 3.2, the emphasis changes to abstracting operations, an an abstract analog of unification is defined. With the addition of straightforward renaming machinery, the abstract unifier can be bolted into the generator framework.

3.1 Abstracting substitutions

As in [15], abstract substitutions are formulated in terms of sharing groups [7, 16] which represent which program variables share variables. Formally, an abstract substitution is structured as a set of sharing groups where a sharing group is a (possibly empty) set of program variables. The intuition is that a sharing group records which program variables are bound to terms that share a variable. Freeness is represented by simply pairing the sharing component with the free program variables. The precise notion of abstraction is first defined for a single substitution via sh and fr and then, by lifting sh and fr, generalised to sets of substitutions.

Definition 6 (*occ*, *sh* **and** *fr*) *The abstraction mappings* $occ : Uvar \times Sub \rightarrow \wp(Pvar)$, $sh : Sub/\approx \rightarrow \wp(\wp(Pvar))$ *and* $fr : Sub/\approx \rightarrow \wp(Pvar)$ *are defined by:*

$$occ(u, \phi) = \{v \in Pvar \mid u \in var(\phi(v))\}$$

$$sh([\phi]_{\approx}) = \{occ(u, \phi) \mid u \in Uvar\}, \quad fr([\phi]_{\approx}) = \{v \in Pvar \mid \phi(v) \in Uvar\}$$

Sub is the set of substitutions; $Sub/\approx$ is Sub factored by renaming $\approx$; $Uvar$ is the universe of all variables; and $Pvar$ is a finite set of program variables.

The abstract domain, $Sub^{\#}_{Pvar}$, is virtually defined by the codomains of sh and fr. Equipping $Sub^{\#}_{Pvar}$ with a pair-wise subset ordering, by virtue of the finiteness of $Pvar$, defines a finite lattice.

Definition 7 ($Sub^{\#}_{Pvar}$ $(\sqsubseteq)$) *The abstract domain,* $Sub^{\#}_{Pvar}$ $(\sqsubseteq)$, *is defined by:* $Sub^{\#}_{Pvar} = \wp(\wp(Pvar)) \times \wp(Pvar)$ *with* $\sqsubseteq$ *defined by:* $\langle \phi^{\#}_{sh}, \phi^{\#}_{fr} \rangle \sqsubseteq \langle \varphi^{\#}_{sh}, \varphi^{\#}_{fr} \rangle$ *if and only if* $\phi^{\#}_{sh} \subseteq \varphi^{\#}_{sh}$ *and* $\varphi^{\#}_{fr} \subseteq \phi^{\#}_{fr}$.

The sh and fr naturally lift to sets of substitutions to finally define α and γ.

Definition 8 (α **and** γ) *The abstraction and concretisation mappings* $\alpha : \wp(Sub/\approx) \rightarrow Sub^{\#}_{Pvar}$ *and* $\gamma : Sub^{\#}_{Pvar} \rightarrow \wp(Sub/\approx)$ *are defined by:*

$$\alpha(\Phi) = \langle \cup_{[\phi]_{\approx} \in \Phi} sh([\phi]_{\approx}), \cap_{[\phi]_{\approx} \in \Phi} fr([\phi]_{\approx}) \rangle$$

$$\gamma(\phi^{\#}) = \{[\phi]_{\approx} \in Sub/\approx \mid \langle sh([\phi]_{\approx}), fr([\phi]_{\approx}) \rangle \sqsubseteq \phi^{\#}\}$$

3.2 Abstracting unification

Unification is abstracted by tracing the steps of a unification algorithm [12]. The unification algorithm, for succinctness, is described in terms of a predicate mgu which basically simplifies and solves finite sets of equality constraints $(E \in)\, Eqn$ of the form $a = b$ where a and b are terms or atoms.

Definition 9 (mgu) *The relation* $mgu \subseteq Eqn \times Sub$ *is defined by:*

$$\begin{array}{r}
mgu(\emptyset, \epsilon) \\
mgu(\{v = v\} \cup E, \zeta) \text{ if } mgu(E, \zeta) \\
mgu(\{t = v\} \cup E, \zeta) \text{ if } mgu(\{v = t\} \cup E, \zeta) \\
mgu(\{v = t\} \cup E, \zeta \circ \eta) \text{ if } mgu(\eta(E), \zeta) \wedge v \notin var(t) \wedge \eta = \{v \mapsto t\} \\
mgu(\{f(t_1, \ldots, t_n) = f(t'_1, \ldots, t'_n)\} \cup E, \zeta) \text{ if } mgu(\{t_1 = t'_1, \ldots, t_n = t'_n\} \cup E, \zeta)
\end{array}$$

The predicate mgu, in turn, defines the set of most general unifiers for E by: $mgu(E) = \{\phi \mid mgu(E, \phi)\}$.

With the aid of a mapping $mgu^{\#} : Pvar \times Term \times Sub^{\#}_{Pvar} \rightarrow Sub^{\#}_{Pvar}$ for solving simplified equations of the form $v = t$, an abstract unification algorithm is straightforwardly constructed by mimicking the recursive simplification steps of mgu. To spare the need to define an extra (composition) operator, however, $mgu^{\#}(v, t, \phi^{\#})$ is defined to abstract a slight variant of mgu. Specifically, if $\varphi \in mgu(\{\phi(v) = \phi(t)\})$ and $[\phi]_{\approx} \in \gamma(\phi^{\#})$ then $mgu^{\#}(v, t, \phi^{\#})$ abstracts the composition $\varphi \circ \phi$ (rather than φ), that is, $[\varphi \circ \phi]_{\approx} \in \gamma(mgu^{\#}(v, t, \phi^{\#}))$.

Definition 10 (*$mgu*$*) *The relation $mgu* : Eqn \times Sub^{\#}_{Pvar} \times Sub^{\#}_{Pvar}$ is defined by:*

$$
\begin{array}{r}
mgu*(\emptyset, \phi*, \phi*) \\
mgu*(\{v = v\} \cup E, \phi*, \varphi*) \text{ if } mgu*(E, \phi*, \varphi*) \\
mgu*(\{t = v\} \cup E, \phi*, \varphi*) \text{ if } mgu*(\{v = t\} \cup E, \phi*, \varphi*) \\
mgu*(\{v = t\} \cup E, \phi*, \varphi*) \text{ if } mgu*(E, mgu*(v, t, \phi*), \varphi*) \wedge v \notin var(t) \\
mgu*(\{f(t_1, \ldots, t_n) = f(t'_1, \ldots, t'_n)\} \cup E, \phi*, \varphi*) \text{ if } mgu*(\{t_1 = t'_1, \ldots, t_n = t'_n\} \cup E, \phi*, \varphi*)
\end{array}
$$

To define the mapping $mgu*$ (and thus the relation $mgu*$) a number of standard auxiliary operators are required [7, 16]. First, $rel(t, \phi*)$ represents the sharing groups of $\phi*$ which are relevant to the term t, that is, those sharing groups of $\phi*$ which share variables with t. Second, in the absence of useful freeness information worst-case aliasing is assumed. Thus, as in [7, 16], a closure under union operator, *, is employed to enumerate all the possible sharing groups that can possibly arise in unification. Third, to succinctly define $mgu*$, it is convenient to lift $\cup$ to sets of sharing groups with $\square$.

Definition 11 (*rel*, * [7, 16] and $\square$)

$$rel(t, \phi^{\#}_{sh}) = \{U \in \phi^{\#}_{sh} \mid U \cap var(t) \neq \emptyset\}$$

$$\phi^{\#\,\star}_{sh} = \phi^{\#}_{sh} \cup \{U \cup U' \mid U, U' \in \phi^{\#\,\star}_{sh}\}, \quad \phi^{\#}_{sh} \square \phi'^{\#}_{sh} = \{U \cup U' \mid U \in \phi^{\#}_{sh} \wedge U' \in \phi'^{\#}_{sh}\}$$

The mapping $mgu*$ is defined in terms of a pair of mappings $mgu^{\#}_{sh}$ and $mgu^{\#}_{fr}$ which apply different analysis strategies according to the freeness of $\phi(v)$ and $\phi(t)$ for $[\phi]_{\approx} \in \gamma(\phi*)$. The default strategy corresponds of $mgu^{\#}_{sh}$ to the standard treatment in the abstract solver *amgu* of [7].

Definition 12 (*$mgu*$*)

$$mgu*(v, t, \phi*) = \langle mgu^{\#}_{sh}(v, t, \phi*), mgu^{\#}_{fr}(v, t, \phi*)\rangle$$

$$mgu^{\#}_{sh}(v, t, \phi*) = \begin{cases} \psi^{\#}_{sh} \cup (rel(v, \phi^{\#}_{sh}) \ \square\, rel(t, \phi^{\#}_{sh}) \ ) \text{ if } v \in \phi^{\#}_{fr} \vee t \in \phi^{\#}_{fr} \\ \psi^{\#}_{sh} \cup (rel(v, \phi^{\#}_{sh})^\star \square\, rel(t, \phi^{\#}_{sh})^\star) \text{ otherwise} \end{cases}$$

$$mgu^{\#}_{fr}(v, t, \phi*) = \begin{cases} \phi^{\#}_{fr} & \text{if } \quad v \in \phi^{\#}_{fr} \wedge t \in \phi^{\#}_{fr} \\ \phi^{\#}_{fr} \setminus var(rel(v, \phi^{\#}_{sh})) & \text{else if } v \in \phi^{\#}_{fr} \\ \phi^{\#}_{fr} \setminus var(rel(t, \phi^{\#}_{sh})) & \text{else if } t \in \phi^{\#}_{fr} \\ \phi^{\#}_{fr} \setminus var(rel(v, \phi^{\#}_{sh}) \cup rel(t, \phi^{\#}_{sh})) & \text{otherwise} \end{cases}$$

$$\text{where } \psi^{\#}_{sh} = \phi^{\#}_{sh} \setminus (rel(v, \phi^{\#}_{sh}) \cup rel(t, \phi^{\#}_{sh}))$$

Note that $rel(v, \phi^{\#}_{sh}) \square\, rel(t, \phi^{\#}_{sh}) = \emptyset$ and $rel(v, \phi^{\#}_{sh})^\star \square\, rel(t, \phi^{\#}_{sh})^\star = \emptyset$ if $rel(v, \phi^{\#}_{sh}) = \emptyset$. Thus, in clause 1 of $mgu^{\#}_{sh}$, $rel(t, \phi^{\#}_{sh})$ need not be calculated if $rel(v, \phi^{\#}_{sh}) = \emptyset$ and similarly in clause 2, $rel(t, \phi^{\#}_{sh})$ need not be computed or closed under union if $rel(v, \phi^{\#}_{sh}) = \emptyset$. Analogous refinements follow if $rel(t, \phi^{\#}_{sh}) = \emptyset$. The correctness of the mapping $mgu*$ is asserted in lemma 2.

Lemma 2

$$[\phi]_{\approx} \in \gamma(\phi^{\#}) \wedge \varphi \in mgu(\{\phi(v) = \phi(t)\}) \wedge \\ \{v\} \cup var(t) \subseteq Pvar \wedge v \notin var(t) \Rightarrow [\varphi \circ \phi]_{\approx} \in \gamma(mgu^{\#}(v, t, \phi^{\#}))$$

The correctness of the relation $mge^{\#}$ follows from lemma 2 and is stated as theorem 5.

Theorem 5

$$[\phi]_{\approx} \in \gamma(\phi^{\#}) \wedge \varphi \in mgu(\phi(E)) \wedge \\ mgu^{\#}(E, \phi^{\#}, \mu^{\#}) \wedge var(E) \subseteq Pvar \Rightarrow [\varphi \circ \phi]_{\approx} \in \gamma(\mu^{\#})$$

Example 1 illustrates that the simplicity of the freeness analysis is not gained at the expense of precision. Indeed the analysis seems to possess much of the power of the freeness analysis of [15].

Example 1 *For the sake of comparison with the freeness analysis of [15], consider the calculation of* $mge^{\#}(\{a = b\}, \phi^{\#}, \psi^{\#})$ *where* $a = p(x_1, x_1, a, x_2, x_2, f(x_1, x_3))$, $b = p(f(x_1, x_2), f(x_3, x_4), x_3, x_5, x_6, x_6)$ *and* $\phi^{\#} = \langle\{\emptyset, \{x_1\}, \{x_2\}, \{x_3\}, \{y_1, y_2\}, \{y_2\}, \{y_3\}, \{y_5\}, \{y_6\}\}, \{x_1, x_2, x_3, y_1, y_3, y_5, y_6\}\rangle$. *The computation reduces to* $mge^{\#}(E', \phi^{\#}, \psi^{\#})$ *where* $E' = \{x_1 = f(y_1, y_2), x_1 = f(y_3, y_4), y_3 = a, y_5 = x_2, y_6 = x_2, y_6 = f(x_1, x_3)\}$. *Therefore, putting* $\phi_1^{\#} = \phi^{\#}$, *and considering each simplified equation in turn,*

$$\begin{aligned}
\phi_1^{\#} &= &&= \langle\{\emptyset, \{x_1\}, \{x_2\}, \{x_3\}, \{y_1, y_2\}, \{y_2\}, \{y_3\}, \{y_5\}, \{y_6\}\}, \\
& && \quad \{x_1, x_2, x_3, y_1, y_3, y_5, y_6\}\rangle \\
\phi_2^{\#} &= mgu^{\#}(x_1, f(y_1, y_2), &\phi_1^{\#}) &= \langle\{\emptyset, \{x_1, y_1, y_2\}, \{x_1, y_2\}, \{x_2\}, \{x_3\}, \{y_3\}, \{y_5\}, \{y_6\}\}, \\
& && \quad \{x_2, x_3, y_1, y_3, y_5, y_6\}\rangle \\
\phi_3^{\#} &= mgu^{\#}(x_1, f(y_3, y_4), &\phi_2^{\#}) &= \langle\{\emptyset, \{x_1, y_1, y_2, y_3\}, \{x_1, y_2, y_3\}, \{x_2\}, \{x_3\}, \{y_5\}, \{y_6\}\}, \\
& && \quad \{x_2, x_3, y_5, y_6\}\rangle \\
\phi_4^{\#} &= mgu^{\#}(y_3, a, &\phi_3^{\#}) &= \langle\{\emptyset, \{x_2\}, \{x_3\}, \{y_5\}, \{y_6\}\}, \{x_2, x_3, y_5, y_6\}\rangle \\
\phi_5^{\#} &= mgu^{\#}(y_5, x_2, &\phi_4^{\#}) &= \langle\{\emptyset, \{x_2, y_5\}, \{x_3\}, \{y_6\}\}, \{x_2, x_3, y_5, y_6\}\rangle \\
\phi_6^{\#} &= mgu^{\#}(y_6, x_2, &\phi_5^{\#}) &= \langle\{\emptyset, \{x_2, y_5, y_6\}, \{x_3\}\}, \{x_2, x_3, y_5, y_6\}\rangle \\
\phi_7^{\#} &= mgu^{\#}(y_6, f(x_1, x_3), &\phi_6^{\#}) &= \langle\{\emptyset, \{x_2, x_3, y_5, y_6\}\}, \{x_3\}\rangle
\end{aligned}$$

Therefore $\psi^{\#} = \langle\{\emptyset, \{x_2, x_3, y_5, y_6\}\}, \{x_3\}\rangle$. *The freeness analysis of [15] (modulo projection) likewise infers that only* x_2 *and* x_3 *are aliased and that* x_3 *is free.*

4 Implementation

The generator semantics constructs a subset of the extensions of an initial goal tree, consisting of generator trees. The set of generators, however, can be an indefinitely large set of indefinitely large trees, and therefore require abstraction. In addition, the cost of applying β in the definition of $\mathcal{O}_g$, without abstraction, can be expensive.

Abstraction of generators is performed in two stages. First, the set of generators is mapped onto a finite directed graph. The nodes of the graph are (labelled with) sets of generators which have common ancestors. (The motivation is to collapse together generators which are likely to have similar call structures and therefore similar data-dependencies.) There is an edge from one node to another if generators at the former have immediate extensions at the latter. No information is lost in this mapping and the collecting semantics defined in terms of this graph can be show to be equivalent to the generator semantics. Second, the sets the generators are represented by (finite) abstract generators.

Interestingly, two problems arise in the generator approach which do not occur in traditional sequential abstract interpretation frameworks. First, to extend a generator involves computing minimal sets of generators. A naive approach involves enumerating all possible subsets of the set of generators. To make the analysis tractable, instead, freeness and sharing information is used to identify generators which may provide bindings for the variables of a sub-goal that definitely need to be bound. The generators which may contribute to binding sub-goal variable are themselves approximated and the resulting abstract tree is unified with the generator being extended. Second, variables cannot be restricted to clauses (as in [1]) since data-dependencies must be considered at the level of the goal. Traditional framework (widening) technology can, however, be adapted to safely reason about recursion.

An implementation of a prototype abstract interpreter based on the generator approach is currently in its later stages of development. The prototype is written in Prolog to simplify profiling and the gathering of cumulative timing information. The rationale for the prototype was to explore whether or not the generator approach is feasible for non-trivial programs and to gather statistical information about the costs of analysing concurrency. One particular aim is to understand the relationship between the freeness component of the analyser and the framework component of the analyser for a variety of freeness analyses. To date, however, only one freeness analysis has been completely implemented.

With the freeness analysis described in this paper, running the compiled analyser (with profiledcode) on a 16M SPARCstation 1 under SICStus 2.1, the interpreter can analyse a test suite of 6 small (less than 10 clauses) kl1 programs in less than 20s total. To date, however, the only medium sized benchmark which has been run through the analyser: matrix.kl1 taken from the Tick benchmark suite. The and-parallelism implicit in matrix.kl1 means that the program would be expensive to analyse by enumeration but with the generator approach, data-dependencies can be extracted from matrix.kl1 in 98s with the prototype.

5 Conclusions

Generators are a promising way for reducing the time complexity of the abstract interpretation of concurrent logic programs. Generators deduce data-dependencies between branches of the proof tree by inferring, for a branch, the set of the branches which must be executed before it. Generators, in addition, trace sharing behaviour. Parameterised by an efficient freeness analysis, the prototype abstract interpreter can deduce ordering information reasonably quickly. This suggests that generators may provide an useful pathway for optimising concurrent logic programs.

Acknowledgements

This work was supported, in part, by ESPRIT project (6707) "ParForce" and SERC grant GR/H15042.

References

[1] M. Bruynooghe. A practical framework for the abstract interpretation of logic programs. *J. Logic Programming*, 10:91–124, 1991.

[2] M. Codish, D. Dams, and E. Yardeni. Derivation and safety of an abstract unification algorithm for groundness and aliasing analysis. In *ICLP'91*, pages 79–93, Paris, France, 1991. MIT Press.

[3] A. Cortesi and G. Filé. Abstract interpretation of logic programs: an abstract domain for groundness, sharing, freeness and compoundness analysis. In *PEPM'91*, pages 52–61. ACM Press, 1991.

[4] S. K. Debray. Static inference of modes and data dependencies in logic programs. *ACM TOPLAS*, 11(3):418–450, July 1989.

[5] I. Foster and W. Winsborough. Copy avoidance through compile-time analysis and local reuse. In *in ILPS'91*. MIT Press, (1991).

[6] S. Gregory. *Parallel Logic Programming in Parlog, The Language and its Implementation.* Addison-Wesley, (1987).

[7] D. Jacobs and A. Langen. Static Analysis of Logic Programs. *J. Logic Programming*, pages 154–314, 1992.

[8] R.S. Kemp and G.A. Ringwood. A generator semantics for producer-consumer analysis. Technical Report 652, QMW, London University, UK, November 1993.

[9] A. King and P. Soper. Ordering optimisations for concurrent logic programs. In *in LFCSS'92*, Tver, Russia, 1992. Springer-Verlag.

[10] A. King and P. Soper. Schedule analysis of concurrent logic programs. In *in JICSLP'92*, Washington, 1992. MIT Press.

[11] A. King and P. Soper. Serialisation analysis of concurrent logic programs. In *in ICALP'92*, Pisa, Italy, 1992. Springer-Verlag.

[12] J. Lassez, M. J. Maher, and K. Marriott. *Foundations of Deductive Databases and Logic Programming*, chapter Unification Revisited. Morgan Kaufmann, 1987.

[13] B. C. Massey and E. Tick. The diadorra principle: Efficient execution of fine-grain concurrent languages. Technical report, Dept. of Computer Science, University of Oregon, 1993.

[14] B. C. Massey and E. Tick. Sequentialisation of parallel logic programs with mode analysis. In *in ICLPAR'93*, 1993.

[15] K. Muthukumar and M. Hermenegildo. Combined determination of sharing and freeness of program variables through abstract interpretation. In *ICLP'91*, pages 49–63, Paris, France, 1991. MIT Press.

[16] K. Muthukumar and M. Hermenegildo. Compile-time derivation of variable dependency through abstract interpretation. *J. of Logic Programming*, pages 315–437, 1992.

[17] J.C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. In *Machine Intelligence 5*, pages 135–151. Elsevier, New York, 1970.

[18] R. Sundararajan and J. Conery. An abstract interpretation scheme for groundness, freeness, and sharing analysis of logic programs. In *12th FST and TCS Conference*, New Delhi, India, December 1992. Springer-Verlag.

[19] R. Sundararajan, A. V. S. Sastry, and E. Tick. Variable threadedness analysis for concurrent logic programs. In *in JICSLP'92*, pages 493–508, 1992.

[20] H. Xia. *Analyzing Data Dependencies, Detecting And-Parallelism and Optimizing Backtracking in Prolog Programs.* PhD thesis, University of Berlin, April 1989.

A Parallel Copying Garbage Collection Scheme for Shared-memory Multiprocessors

Extended Abstract

Khayri A. M. Ali
Swedish Institute of Computer Science, SICS
Box 1263, S-164 28 Kista, Sweden
khayri@sics.se

1 Introduction

Today many parallel computers are commercially available and a lot of researchers try to utilize this opportunity to speed up the execution of programs. A number of functional, logic, and object-oriented programming environments have been implemented on different parallel architectures. The shared-memory multiprocessors are the ones most used ones as test beds. A common problem in realizing these programming environments on such architectures is how to efficiently manage the heap storage using automatic garbage collection (GC). These languages are based on dynamic object creation and automatic reclamation of storage during computation. A garbage collector retains a program's data that is in use (*active*) and reclaims data that is unreachable by the program (*garbage*). Garbage collection is necessary since even large (*virtual*) memory is finite. In addition to reclaiming storage, a garbage collector can also restore *locality* to fragmented data by dynamically compacting it, thereby improving the performance of caches and virtual memory systems and reducing memory requirements.

Copying GC is very attractive for systems that have a high rate of garbage generation as in functional programing and the family of concurrent logic programming. This is because in the copying method garbage collection time is proportional to the amount of data in use by the system. This is in contrast to the mark-and-sweep method, which has garbage collection time proportional to the entire heap area. The reference-counting method avoids this problem by incremental collection, but it has a problem with memory compaction which does not exist in the other two methods. The main drawbacks of the copying method are that 1) twice as much heap memory is needed as in the other alternatives, and 2) collection is not done in real-time. However, for the above mentioned environments the copying method has been widely used for its high efficiency in comparison with the other alternatives.

The basic sequential copying collector [1] requires two memory spaces of equal size. When one memory space has been exhausted, active data structures will be copied to the other memory space. This copying process usually traverses active data in a breadth-first manner.

Section 2 gives the two earlier schemes that have been proposed for parallelization of the basic sequential copying collector with breadth-first traversal of active data. Section 3 describes a sequential copying collector based on traversal of active data in a depth-first manner and some of its advantages over a copying GC scheme based on breadth-first. Section 4 gives the idea of how to parallelize the scheme presented in section 3.

2 Earlier Work

Two earlier approaches [2, 3] have proposed parallelization of the basic copying GC scheme for shared-memory machines. The Halsted approach [2] is based on statically dividing the entire heap area equally between the PEs, and each PE copying active objects into its private new heap area. Thus it does not support load-balancing among the PEs. The Imai and Tick approach [3] organizes the global heap area in such a way that objects of equal size are allocated in the same memory block, and any PE can garbage collect objects in any memory block according to a dynamic load-balancing scheme. It uses a global stack for saving GC work that can be taken by any processor in the system. Memory organization used by the latter approach [3] does not improve locality of reference unlike the former one [2] whereas the latter approach allows dynamic load-balancing.

In the two schemes, the whole heap space is assumed to be one contiguous area and the active data is traversed in a breadth-first manner.

Our scheme is a parallelization of a sequential copying GC scheme based on traversing active data in a depth-first manner. It improves locality of reference, supports dynamic load-balancing, and is also suitable for parallel copying of structure terms.

3 The Sequential Garbage Collection Scheme

The main two advantages of a scheme based on traversing active data depth first over the corresponding one based on breadth-first traversal are:

- For a storage heap consisting of a number of non-contiguous memory blocks, the depth-first scheme does not require an extra mechanism for maintaining the cells to be scanned in different memory blocks. The non-contiguous memory blocks organization of the storage heap is much more flexible than the one based on one large contiguous heap area. This is because it is much easier to get a number of smaller memory blocks than to get one big block from any operating system

- For some applications, like ours, it is more efficient and simpler during the GC process to finish garbage collection of certain objects before starting with another one. In this case, the depth-first scheme is much more straightforward than the breadth-first scheme. The depth-first scheme maintains cells to be scanned in a chain whereas the breadth-first scheme has to use an extra mechanism for achieving the same result. In the depth-first scheme, there is therefore no extra cost for mixing pointer cells and arbitrary data on the heap.

The storage heap is represented by a number of non-contiguous memory blocks that are maintained in two linked lists, one list for memory blocks containing data allocated in the heap and another for free memory blocks. When the ratio of used memory blocks to free memory blocks reaches some specific threshold, the garbage collector is invoked. The collector copies live data from the used memory blocks to free blocks, and then returns the former blocks to the list of free memory blocks. This copying process traverses each copied data structure depth first. That is, if the current cell to be investigated in a copied structure A points to uncopied structure B, the next cell to be investigated will be the last cell of a copy of B. The next cell of A will be investigated only when all cells of B and the new copied structure accessible from B have been completely investigated. That is, cells are investigated in last-copied-first-investigated manner.

Two pointers, L and P, and a tag register, T, are used to maintain the chains of cells to be investigated. Moreover one bit of each cell is used by the garbage collector for marking. There are two uses for the mark-bit: one for marking already copied data and the other for marking the last cell to be scanned in each copied structure. Before GC, all cells are unmarked. When a structure is copied, the first cell of the old copy is marked and a pointer to the new location (*forward pointer*) overwritten in the same cell. Cells to be investigated are divided into two chains: one chain links cells to be investigated in the current structure and the other links the remaining cells to be investigated. L is used to point to the beginning of the first chain and P to the beginning of the other chain. The first chain ends with a marked cell and the other chain ends with a *dummy cell.*

For simplicity, let us assume a term is a *structure*, which is represented by a tagged pointer (STR, pointer) pointing to an area of $n+1$ contiguous cells where n is the arity and the first cell contains $functor/n$.

Initially, L points to the root structure, P to a *dummy cell*, and T is a *dummy tag*.

If L points to an uncopied structure, the structure is copied and the first cell of the old structure is marked and a forward pointer overwritten in it. The first argument cell of the copied structure is also marked to indicate the last cell to be investigated in the structure. The contents of P and T will be stored in the investigated cell, the tag of the investigated cell is saved in T, P points to the investigated cell, and L points to the last cell of the copied structure.

If L points to a copied structure, the new location will be stored in the current cell and L is advanced to point to the next cell. The next cell will be in the current structure or the one pointed to by P depending on whether the current cell is or is not marked respectively. In the latter case, a pointer to the current structure is calculated from L and T. The new value of L is P, and the new values of P and T are restored from the cell pointed to by L.

The GC is completed when P points again to the *dummy cell*

4 The Parallel Garbage Collection Scheme

In the sequential scheme, cells to be investigated are maintained in two chains: one for cells in the current investigated structure and the other for all remaining cells to be investigated. The idea of parallelizing this process is to divide the latter chain into a number of smaller chains and to make them available to the other processors (*workers*). Each chain is a piece of work that can be processed by any worker. Each worker maintains its chains in its own *load* stack. Idle workers during GC can get work from other workers with excess load by just getting a small amount of information (only two words) describing the chain of cells to be investigated.

In the assumed parallel environment, the storage heap is represented by one global pool of free memory blocks and a number of used memory blocks associated with each worker. During computation workers get free memory blocks from the global pool until the ratio of used blocks to free blocks reaches a certain threshold, whereupon the parallel garbage collector will be invoked. It stops computation in the system, then each worker, starting from its root structure, copies accessible data into free blocks from the global pool. During this process, each worker maintains its excess work in its own load stack. When a worker finishes its current work and all work in its load stack, it will try to find work in that of another worker. Workers distribute work between themselves until all workers become idle. In this situation, every worker returns its old used memory blocks to the global pool, completing a GC cycle.

In the full paper we are going to answer to the following questions:

- How to divide a chain of cells into smaller chains?
- How to represent each chain with minimal space overhead?
- How to distribute work among workers with dynamic load-balancing?
- How to process a non-local chain?
- How to support locking of cells?
- How to use the scheme to copy structure terms in parallel?

5 Conclusions

The paper presents parallelization of a sequential copying GC scheme with depth-first traversal of active data. The scheme is suitable for non-contiguous memory blocks of the storage heap in shared-memory multiprocessors. It supports dynamic load balancing. The space requirement for the scheme is very small; a small stack is associated with each processor. The extra processing power is due to locking of objects in the old space. This is the essential overhead in any parallel scheme in comparison with the corresponding sequential ones. The scheme is also suitable for parallel copying of structure terms.

References

[1] C. J. Cheney. A Non-recursive List Compacting Algorithm. *Communications of the ACM*, 13(11): 677–678, November 1970.

[2] R. H. Halstead. Implementation of Multilisp: Lisp on a Multiprocessor. *ACM Symposium on LISP and Functional Programming*, Austin, Texas, pages 9–17, 1984.

[3] Imai and E. Tick's paper. 1993.

An And/Or-Parallel Implementation of AKL (DRAFT)

Johan Montelius, Khayri M. Ali
{ jm, khayri@sics.se }

March 2, 1994

Abstract

The Agents Kernel Language (AKL) is a general purpose concurrent constraint language. It combines the programming paradigms of search-oriented languages such as Prolog and process-oriented languages such as GHC [9].

The paper is focused on three essential issues of parallel implementation of AKL on shared-memory multiprocessors: how to maintain multiple constraint windows, how to represent the guard computation and how to distribute work among workers.

A simple scheme is used for maintaining multiple constraint windows. A worker will immediately see conditional constraints placed on variables. All workers will have a coherent view of the constraint stores. A locking scheme is used that entails very little overhead for operations on local variables. Local variables can be bound in one atomic exchange operation.

Continuations are used to represent sequences of untried goals. There is very little overhead for deterministic non-suspending agents. One advantage of continuations over goal stacking is that the granularity of work becomes larger.

The computation state is represented as a shared data structure among the workers, and each worker maintains pointers to its own set of And/Or tasks. The tasks are distributed by just moving pointers from one worker to another. By this way hot-spots are avoided.

1 Introduction

To be written.

2 The Configuration

An AKL configuration is a tree that consists of *choice-boxes* and *and-boxes*. Each choice-box represents an agent and each and-box represents a guard computation. An and-box is linked to its child choice-boxes, its sibling and-boxes and its parent choice-box. A choice-box is in a similar way linked to its child and-boxes, sibling choice-boxes and parent and-box. The root choice-box has a no parent and-box. All sequences of and-boxes and choice-boxes must be ordered; and-boxes to implement the conditional guard and choice boxes to implement choice splitting.

Each variable is local to an and-box, the and-box is called the *home* of the variable. A constraint on a variable in the home of the variable is an unconditional constraint. A constraint on a variable external to the and-box in which the constraint is made is a conditional constraint.

The constraints in an and-box form a *constraint store*. A constraint in an and-box is either: consistent and entailed by, consistent but not entailed by or inconsistent with the constraints in the constraint stores above the and-box.

An and-box is *stable* if no deterministic step can be taken and no constraint store below the and-box contains a non-entailed constraint on a variable external to the and-box.

The constraints are here limited to equality on rational trees, constraints on variables will be referred to as bindings. A full description of the computation model of AKL semantics can be found in [5].

3 Representation of the Configuration

A worker is a process that performs rewrite operations on the configuration. The execution state of a worker consists of a position in the configuration. The and-box in which the worker is positioned in is called the current and-box and the constraint store of that and-box is called the current store. This section describes the main parts representing the configuration.

and-boxes

An and-box contains an *environment identifier*, a *hanger*, and a list of bindings. The environment identifier is a cell used by variables to identify the *home* and-box. The hanger is an indirection to allow deallocation while there remain suspension references.

An and-box holds a single linked list of insertion points. Each point is either dead or referring to a choice-box.

and-continuations

Untried goals in a guard are represented by an and-continuation. The continuation contains the permanent registers, a code pointer, a pointer to the next and-continuation and the insertion point of the continuation.

choice-boxes

A choice-box contains a choice-continuation, ...

terms

Variables are either unconstrained or constrained. An unconstrained variable has apart from its tag an environment identifier. Constrained variables hold a reference to a structure holding the environment identifier and a list of suspensions.

A compound term contains, apart from its arguments, an environment identifier. The environment identifier is used when parts of the configuration are copied during a choice split.

environments

The environment identifier is a cell which contains a reference to the home and-box or a forward pointer to another environment identifier. If an and-box is promoted a forward pointer is placed in its environment cell. To check whether a term is local the environment identifier of the term is compared with the environment identifier of the and-box. If these are equal the term is local, otherwise the environment cell might have to be dereferenced. This scheme is similar to the scheme designed in [7].

4 Binding Window

There exist several solutions to the problem of maintaining multiple binding environments [3]. These solutions are however designed for or-parallel systems where different bindings exist in different or-branches in an execution state. In the proposed abstract machine or-nodes are not used; alternative environments are created by explicitly copying all structures involved. There is only need to maintain bindings on different levels in the configuration.

Since the number of levels in an AKL configuration is normally far less than the levels of or-nodes in a Prolog execution, a very simple scheme can be used. The scheme is close to the computation model and places little overhead on the most frequent operations.

binding lists

Unconditional bindings can never be removed and can therefore be recorded in place. Conditional bindings must only be visible in or below the and-box in which the binding occurs and are therefore recorded in the binding list of the and-box.

A binding list consists of an entry for each bound external variable. An entry contains, apart from a reference to the variable, the binding of the variable or a list of suspensions. An entry that contains a list of suspensions is called a *suspension entry*. The suspension entry does not constrain the variable, its purpose is only to keep track of suspensions. An entry that is not a suspension entry is a *proper entry*.

A suspension of a constrained variable is referring to an and-box immediately below the home of the variable. This and-box either contains a proper entry for the variable or a suspension entry with suspensions referring to and-boxes immediately below the and-box. The hierarchical structure of the suspensions is used both to keep locality of suspensions and to detect stability.

To find the current binding of an unbound external variable each binding list from the current and-box to (but not including) the home of the variable must be examined. If no entry is found the variable is unbound.

bindings

A local variable is bound in place. If the variable is constrained the suspended and-boxes are woken.

An external variable is bound by first adding the binding to the local constraint store. Secondly a suspension is added to each constraint store from the parent and-box to (but not including) the home of the variable and a suspension on the variable itself.

locking

A local variable needs not to be locked to perform a binding operation. The variable is bound by exchanging its current value with the value of the binding. If the returned value is either an unconstrained or constrained variable the binding succeeded. If the returned value is a term that term must be unified with the binding, the binding can be left in place.

When a new entry is added to a binding list the list must be locked. This is done by exchanging the list pointer with a lock value. If the returned value is a pointer the lock was taken and the operation can proceed.

If a new binding is added the list must be examined to find any existing entry for the variable. If a proper entry is found the new binding is unified with the existing entry. If a suspension entry is found the suspended and-boxes are woken.

If a suspension is added and a proper entry is found the and-box re-executed. If a suspension entry is found the new suspension is added to its list of suspensions otherwise a suspension entry is created.

A suspension on a variable is added by first exchanging its current value with a tagged reference to itself. In this way the access to the variable is locked, any access to the variable will loop on the circular reference until the operation is completed. The returned value is then examined. If a variable is returned the suspension can be added to the variable and the value returned to the variable. If a term is returned: the variable has been bound or is locked. If the term is a tagged reference pointing to the variable itself, the variable is locked and the suspension operation must be performed again. If it is a proper term the variable has been bound and the and-box is scheduled for re-execution.

stability

Stability of an and-box is detected by examining the entries in the binding list. A *living entry* for a variable is either a proper entry for the variable or a suspension entry that refers to a *living and-box* with a living entry for the variable. An and-box is unstable if and only if it holds a living entry.

The last worker to leave the and-box makes the stability check. In this way the check can be made without any locking. Dead entries can be removed in order to keep the representation compact.

choice split

A choice split is performed by finding a candidate and-box, make a copy of the parent and-box where the siblings of the candidate has been removed and remove the candidate from the parent and-box. When the parent and-box is copied all local terms are copied. No external terms are copied. The environment identifier in compound term is used to determine whether a term is local or external.

Terms are copied so that the two and-boxes will not share any local variables. Ground terms need not to be copied. As soon as it is determined that a term is ground (during copy or gc) a null identifier replaces the original identifier. In subsequent copy operations the ground term needs not to be examined.

A copy operation can be performed by a worker independently of the execution in other parts of the configuration. No external structures are modified during the operation.

pros and cons

The main advantage of the described binding scheme is the simplicity of installation and de-installation. A worker can move freely in the configuration since it does not need to update any private information. This allows for fast task switching. The explicit representation of the binding window makes bindings immediately visible to all workers. Inconsistencies are detected at an early stage and all workers have a consistent view of the bindings in the configuration.

A disadvantage is the non-constant time operations. To access or to add a new binding can in the worst case be a very costly operation. In practice it does not cause a problem. The majority of variables accesses are made to local variables, in which the binding is found in place, or to variables that are local to the parent and-box, in which only one binding list is examined. These claims will of course have to be verified by examining several larger AKL programs. The experience gained so far does, however, fully support the assumption.

The scheme for locking places very little overhead on the most frequent operation namely the binding of a local variable. The use of a self referring reference as a lock does not alter the normal dereferencing scheme. The disadvantage is that the environment identifier of a constrained variable is unaccessible during the operation.

The comparison with or-parallel schemes for Prolog [3] is not straightforward. The non-deterministic computation sacrifices constant time task creation whereas the constraint store scheme sacrifices constant time variable access. The criticism in the development of the scheme has been that local variable access should be a constant time operation. In order to achieve this, constant time choice splitting or constant time task switching had to be sacrificed. A copying strategy for choice splitting was chosen since it simplifies the scheme for the constraint store.

5 Guard computations

One way of implementing a concurrent language is to use goal stacking [2, 1] i.e. immediately create representations of the goals in a promoted body. The approach has several advantages in that it provides a uniform execution model. One disadvantage is however that unnecessary work is performed if a goal fails. This is of course not a problem in a flat committed choice language where failure is treated as an exception but can be a problem in a deep language where failure of goals is a normal behaviour.

In WAM [10] the environment is used to represent the remaining goals in a body. It is also used in and-parallel implementations of Prolog such as &-Prolog [4] and DDSWAM [8]. The method has proved very efficient in the sequential implementations of AKL [6] and we believe that it has additional advantages in a parallel implementation. Apart from the advantages of lazy creation of goals it can keep the granularity of work more coarse.

insertion

During the execution of a guard one of two registers is always valid: the *current insertion point* or the *current and-continuation*.

If a goal suspends and the current and-continuation is known, the insertion point of the and-continuation is used as the insertion point of the created choice-box. A new insertion point is created to the right as the new insertion point of the and-continuation. If the current and-continuation is unknown the current insertion point is used as the insertion point of the created choice-box.

If an and-continuation is promoted and the current and-continuation is known, the promoted continuation is linked to the current and-continuation. The promoted and-continuation adopts the insertion point of the current and-continuation and is thereafter the current and-continuation. If the current and-continuation is unknown, the current insertion point is used as the insertion point of the promoted and-continuation.

The "proceed" and "execute" instructions remove the current and-continuation and, if it exists, delegate its insertion point to the next and-continuation and make it the current and-continuation. If the removed and-continuation is the last and-continuation, the insertion point of the removed and-continuation is made the current insertion point. The current and-continuation is then undefined.

A "guard" instruction will, if a promotion is not allowed, make the current and-continuation the body of the and-box.

suspensions

Each suspended and-box uses a *hanger* to avoid multiple references. A suspension will always refer to the hanger of a box. If an and-box fails a null entry is entered in the hanger. Any suspension that refers to the hanger can then be ignored.

synchronization

The proposed scheme allows the sequence of choice-boxes and and-continuations to be maintained without locking operations. A new insertion cell is only created when an agent suspends, the worker then has the only reference to the current and-continuation and may manipulate it and its insertion point without any interference

A disadvantage of the scheme is that the list of insertion points will contain dead entries. The list is, however, only traversed when all goals have been executed and it is to be decided whether the guard is solved or not. Note that dead entries only are generated when goals have suspended. Dead entries can be removed at any point in the execution.

In a non-suspending execution the untried goals will be represented by a sequence of and-continuations. Only one insertion point is needed as promoted and-continuations inherit the current insertion point. It is easy to detect that the guard computation is solved.

6 Tasks

A worker will be driven by tasks. Each level in the configuration will have its associated tasks. All tasks at a given level have to be handled before the worker is allowed to move up a level. The tasks can therefore be stored in stacks.

There are two types of and-tasks: *wake* and *a-cont*. Wake tasks are created when constrained variables are bound. Each suspended *living* and-box will generate a wake task.

A promoted and-continuation will generate an a-cont task. A worker will after a promotion examine its wake stack to find new work. If nothing is found it examines the a-cont stack.

There are three types of or-tasks: *c-cont*, *promote* and *split*. A c-cont task is created on demand ......

load distribution

The task can be distributed among workers to dynamically balance the load. If a worker runs out of tasks it will simply try to find another worker with extra work and signal it for getting some tasks. Since the whole configuration is represented as a shared data structure among the workers, a task is distributed simply by copying a pointer from the giving worker to the receiving worker.

pruning

When a pruning operation is performed by a worker, all tasks in scope of the operation should be pruned as early as possible to reduce wast computations. In order to be able to prune work is being executed by the other workers, the representation of the configuration must provide information indicating the positions of the workers in the configuration. Thus each and-box is associated with a *worker-bitmap* indicating workers at and below it. With help of this information a worker can identify the other workers in scope of the executed pruning operation and signal them to perform pruning.

References

[1] Vitor Santos Costa, David H. D. Warren, and Rong Yang. The Andorra-I engine: A parallel implementation of the Basic Andorra model. Tech-

nical note, University of Bristol, Department of Computer Science, March 1990.

[2] Jim Crammond. Implementation of committed choice logic languages omn shared memory multiprocessors. Phd thesis, Heriot-Watt University, 1988.

[3] Gopal Gupta and Bharat Jayaraman. On criteria for Or-Parallel execution models of logic programs. In Saumya Debray and Manuel Hermenegildo, editors, *Proceedings of the 1990 North American Conference on Logic Programming*, pages 737–756, Austin, 1990. ALP, MIT Press.

[4] M. V. Hermenegildo and K. J. Greene. &-Prolog and its performance: Exploiting independent And-Parallelism. In David H. D. Warren and Péter Szeredi, editors, *Proceedings of the Seventh International Conference on Logic Programming*, pages 253–268, Jerusalem, 1990. The MIT Press.

[5] Sverker Janson and Seif Haridi. Programming paradigms of the Andorra kernel language. In Vijay Saraswat and Kazunori Ueda, editors, *Logic Programming, Proceedings of the 1991 International Symposium*, pages 167–186, San Diego, USA, 1991. The MIT Press.

[6] Sverker Janson and Johan Montelius. The design of the AKL/PS 0.0 prototype implementation of the Andorra Kernel Language. ESPRIT deliverable, EP 2471 (PEPMA), Swedish Institute of Computer Science, 1992.

[7] T. Miyazaki, A. Takeuchi, and T. Chikayama. A sequential implementation of concurrent Prolog based on the shallow binding scheme. In *Symposium on Logic Programming*, pages 110–118. IEEE Computer Society, Technical Committee on Computer Languages, The Computer Society Press, July 1985.

[8] Kish Shen. Implementing dynamic dependent And-Parallelism. In David S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming*, pages 167–183, Budapest, Hungary, 1993. The MIT Press.

[9] Kazunori Ueda. Huraded horn clauses. Tecnical Report TR-103, ICOT, July 1985.

[10] D. H. D. Warren. An abstract prolog instruction set. Technical Report 309, SRI International, 1983.

Hybrid tree search in the Andorra model

Remco Moolenaar and Bart Demoen
Department of Computer Science
Celestijnenlaan 200 A, 3001 Heverlee, Belgium
e-mail: {remco, bimbart}@cs.kuleuven.ac.be

Abstract: In this paper two techniques are described to further limit the amount of nondeterminism in the Andorra model. The first technique is to enhance the nondeterministic step with intelligent selection criteria, instead of doing plain left-most selection. The other technique is to remove nondeterministic alternatives that are inconsistent with the surrounding environment. Implementation results show that for ParAKL [6] the amount of nondeterminism is reduced and stabilized (independent of the order of the goals) by these techniques.

1. Introduction

During the last couple of years several solutions have been proposed to extend the Prolog left-to-right depth-first search strategy. One solution is to introduce the notion of determinism in the search strategy itself. The Andorra Model [11] uses this notion to delay construction of nondeterministic parts of the computation tree until all deterministic computations are finished. The major advantage of this model is that it decreases the amount of nondeterminism of a logic program and thus improves overall efficiency, because backtracking search is inefficient, compared to forward deterministic execution [5].

One disadvantage of the Andorra Model is the use of the Prolog left-to-right depth-first selection of nondeterministic goals if no deterministic computations are available. In other words, the Andorra Model improves deterministic computation, but keeps the old-style Prolog search order.

One way to improve this, is to change the order of goals during backward execution [1]. Goals that would probably fail again are moved to the left part of the computation tree. This increases the priority of this goal at nondeterministic goal selection.

In this paper, we describe another approach: instead of using the left-to-right depth-first selection for nondeterministic goals, an intelligent, heuristic based selection mechanism is used, using some analogous techniques known from database systems and constraint logic programs.

On top of this approach, a forward checking algorithm has been implemented to remove alternatives from the selected nondeterministic goal before copying is done. Both techniques are implemented in our parallel implementation of AKL, called ParAKL [6], and the first results (running the system sequentially) show that these techniques improve search in AKL and, in general, in the Andorra Model.

The rest of the paper is organized as follows: section 2 gives a short introduction to AKL, in particular about the copying involved in the non-deterministic promotion step; section 3 gives an overview of the different heuristics that can be used to select a goal; section 4 describes the forward checking algorithm actually implemented and in section 5 and 6 conclusions and plans for future work are formulated.

2. The Andorra Kernel Language.

In this section we present a short introduction of the Andorra Kernel Language (AKL), mainly to define the terminology used in the rest of the paper. For a more detailed review of AKL we refer to [2].

2.1. *The Andorra Kernel Language syntax*

AKL contains the following syntactic categories:

```
   <guarded clause> ::= <head> :- <guard><guardoperator><body>
             <head> ::= <program atom>
     <guard>,<body> ::= <sequence of atoms>
             <atom> ::= <program atom> | <constraint atom>
   <guard operator> ::= '?' | '->' | '|'              (wait, cut, commit)
```

For reasons of simplicity of the explanation of the implementation design described in this paper, the constraint system of Prolog and GHC is used. This means that, a constraint atom is of the form X = t or X = Y and is called a binding. A definition of a predicate is a finite sequence of guarded clauses with the same head atom and the same guard operator, defining the predicate of the head atom. Cut and commit are pruning guards operators. The wait-operator is used for all-solutions search. The body of a guarded clause is evaluated after the completion of the guard and promotion of the clause (see the rules in the next paragraph).

2.2. *The computation model*

The computation model can be captured in a number of rewrite rules, that describe a single execution step: one configuration is rewritten to the next configuration. A configuration is a nested expression built from atoms and boxes. The syntax is defined as follows:

<configuration> ::= <and-box> | <or-box>
<and-box> ::= **and**(<sequence of local goals>)$_{\text{<set of variables>}}$
<or-box> ::= **or**(<sequence of configurations>)
<local goal> ::= <atom> | <choice-box>
<choice-box> ::= **choice**(<sequence of guarded goals>)
<guarded goal> ::= <configuration><guard operator><sequence of atoms>

In the sequel, we use the following conventions:

the letters P, R, S and T denote a possibly empty sequence of local goals
A denotes an atom
B denotes a sequence of atoms
C denotes a sequence of constraint atoms
G denotes a goal and the sequence of atoms in the guard of a guarded clause
the letters V and W denote sets of variables

An and-box is called quiet if it does not restrict variables outside the local variables of the and-box. The conditional rewrite rules are defined as follows.

Rule 1: Local forking rule

$A \Rightarrow$ **choice(and(**G_1**)**v_1 % B_1, …, **and(**G_n**)**v_n % B_n**)**

Condition: H_i :- G_i % B_i defines A.
V_i are local variables. % denotes the guard operator.

Rule 2: Deterministic promotion rule

and(R, **choice(**C_v % B**)**, S**)**$w \Rightarrow$ **and(**R, C, B, S**)**$v \cup w$

Condition: C_v is solved (i.e. is an empty sequence of goals). Moreover, if % is a pruning guard operator, C_v must be quiet.

Rule 3: Cut rule

choice(R, C_v -> B, S**)** $\Rightarrow$ **choice(**R, C_v -> B**)**

Condition: C_v is quiet and solved.

Rule 4: Commit rule

choice(R, C_v | B, S**)** $\Rightarrow$ **choice(**C_v | B**)**

Condition: C_v is quiet and solved.

Rule 5: Nondeterministic promotion rule

and(P, **choice(**R, (C_v? B), S**)**, T**)**$w \Rightarrow$

or(and(P, C, B, T**)**$v \cup w$, **and(**P, **choice(**R, S**)**, T**)**w**)**

Condition: the and-box is stable, i.e. no other rule is applicable to any subgoal of the and-box and there are no bindings in this and-box to variables outside the local variables of the and-box.

Rule 6: Environment synchronization rule

and(R)$w \Rightarrow$ **fail**

Condition: the constraint of R is incompatible with the environment.

Rule 7: Failure propagation rule

and(R, **choice**(), S) $\Rightarrow$ **fail**

Rule 8: Choice elimination rule

choice(R, (**fail** % B), S) $\Rightarrow$ **choice**(R, S)

Rule 9: Guard distribution rule

choice(R, **or**(G, S) % B, T) $\Rightarrow$ **choice**(R, G % B, **or**(S) % B, T)

The next section gives a more detailed description of the nondeterministic promotion rule of AKL, which is necessary to fully describe the improvements made in the next two sections.

2.3. The nondeterministic promotion step

The nondeterministic promotion rule is an essential part of AKL and is necessary to do nondeterministic computation steps (i.e. search).
The difference between AKL and Prolog is that the former uses copying and the latter uses choicepoint creation and backtracking.
This copying can be visualized using an example:

```
:- a(X), b(X).

a(1).
a(2).
a(3).

b(X) :- X > 2 : write(X).
```

The configuration before applying the nondeterministic promotion rule is:

```
and(
    choice( and(X=1), and(X=2), and(X=3) ),
    choice( and(X > 2) : write(X) ) )
```

After applying rule 5, the configuration is as follows:

```
or(
    and( X=1, choice( and(X > 2) : write(X) ) )
    and(
        choice( and(X=2), and(X=3) ),
        choice( and(X > 2) : write(X) ) ) )
```

The surrounding and-box of a/1 is copied (that is, the choice-box of b/1 is copied) and thus creates two independent computations, each exploiting a different part of the search tree.
The following remarks can be made:

- the nondeterministic promotion rule is rather expensive and the amount of copying depends on the size of the surrounding and-box (which can be rather large)
- decreasing the amount of nondeterministic promotion rule applications would increase the overall efficiency of the implementation.

3. Goal selection in the nondeterministic promotion rule

Current implementations of the Andorra Model [3][8] use the left-to-right depth-first rule to select the candidate which is allowed to promote one of its alternatives nondeterministically. This means that these implementations have a Prolog-like behavior for search programs.
But as the order of execution of goals, in the deterministic phase, is not defined, due to the fact that deterministic goals are selected first, there is no good reason to use the old style Prolog selection rule for *nondeterministic* goals. Moreover, insisting on a left-most selection, violates the programming in logic idea where procedural considerations should be of less concern to programmers.
It is quite obvious that criteria used in database systems [9] and constraint logic programming [10] can be used here too. However, because of the different behavior of search in the Andorra Model, some other criteria must be used as well.

3.1. Selection criteria

The selection algorithm is a combination of known techniques [9][10] and specialized criteria. The known techniques, that are included in the algorithm, are the following:

- Select the predicate with the least number of alternatives. This criterion still holds if only one of the alternatives is promoted and the rest is kept in the choice-box, because this goal is still a good candidate after failure of the first promoted alternative.
- First failure principle.
- Select the goal which is the most constrained goal[1], that is, the goal with the largest number of constraints on every argument.

These general techniques are an important part of the selection algorithm, but, because in the Andorra Model, goals with plain variable unification (equality constraints) can be subject to selection as well, an extra selection criterion is necessary to select goals with *useful* bindings (variable-to-atom, etc... instead of variable-to-variable unification). This additional selection criterion prevents the computation to enter some infinite branches as shown by the following example:

```
delete(X, [X|L], L).
delete(X, [Y|L1], [Y|L2]) :- delete(X, L1, L2).

:- delete(X, [1,2,3,4], R), delete(Y, R, R1), constraints(X, Y).
```

If, for instance, the variable 'Y' is more constrained than variable 'X', the criterion based on the constraints, would select the second delete/3 goal. But, as can be seen from the equality constraints stored in the alternatives of the second delete/3 goal[2], no useful binding is available. If the second delete/3 goal is selected systematically, the computation would enter an infinite branch of the computation tree.

The solution to this problem consists in counting the number of known arguments for a candidate goal and to lower the priority for such a candidate if there are variable-to-variable unifications stored in the alternatives. If the number of known arguments of a candidate goal is equal to the arity of this goal then its priority is raised. Such a goal can be seen as a test goal: no local bindings are stored, no computation can be restarted after promotion of an alternative. Such a goal has a big chance to be deterministic[3] and must be selected as soon as possible.

3.2. The selection algorithm

The selection algorithm, which consists of the previous described criteria, is implemented in the sequential version of our parallel AKL system called ParAKL [6]. It is our intension to upgrade the parallel version to include this algorithm as well. For this we need a parallel version of the nondeterministic promotion rule.

1. This is an instantiation of the first failure principle.
2. The first alternative unifies 'Y' with the local variable 'X', 'R' with the list [X|L] and 'R1' with the local variable 'L'. The second alternative unifies 'R' and 'R1' with lists sharing the same head variable 'Y'.
3. The call 'member(1,[1,2])' is deterministic, but a nondeterministic promotion rule must be applied to resolve this.

To be able to efficiently make a selection, all the available candidate goals are saved in a linked list during the deterministic phase of the execution. That is, if a wait guard suspends and there is a branchpoint for this alternative (which is equal to a choice-box if there is more than one alternative) and this branchpoint is not yet a candidate, a reference to this branchpoint is saved in the candidate list.

At application of the nondeterministic promotion rule, the selection algorithm calculates the priorities of all the candidate goals in the candidate list and selects the candidate goal with the highest priority. Then one of the alternatives of this goal is promoted nondeterministically.

The priority of a candidate goal is the sum of two priorities: the first one is based on the known techniques, the second one on the additional selection criterion, which takes into account the unification on and the instantiation of the arguments of a candidate goal.

<u>The first priority: looking at constraints on a variable</u>

The first priority is the sum of the priority calculations for every argument of the candidate goal that is a constrained variable.

The priority of a single argument is calculated as follows (for every suspension on this variable):

- If the suspension is a unification suspension and the branchpoint of the alternative that created the suspension is not equal to the candidate branchpoint: raise the priority of the candidate (another choice-box has a binding for this variable as well).
- If the suspension is a constraint and the number of variables on which this constraint is suspended is equal to one[1]: raise the priority (the nondeterministic promotion of the candidate will solve the constraint).
- If, of all the branchpoints that created a binding for this argument, the branchpoint of the candidate goal has the least number of alternatives: raise its priority (this is calculated when the complete suspension list has been traversed).

The amount of calculation needed for this is limited. It is bound by the arity of the candidate goal times the average number of suspensions on a variable. In practice, this is an upper bound, because only output arguments will be constrained. This will approximately halve the total number of suspensions.

<u>The second priority: looking at the bindings of the arguments</u>

The second priority is calculated as follows: first search for useful bindings made in one of the alternatives of the candidate goal. If none is found or the number of useful bindings is too low, the lowest priority is returned as the result of this calculation.

Otherwise, if all the arguments of the candidate goal are known (i.e. all arguments are bound), raise the priority of the candidate goal.

1. This means: the constraint is suspended on just one variable.

The complexity of this calculation depends on the arity of the candidate goal plus the total number of bindings made in all the alternatives.

The biggest problem is to combine those two calculations and to find the best weights for every step of the calculation. The algorithm that is implemented assigns a weight of 50 to a ground candidate goal, a weight of 20 for every solvable constraint[1], a weight of 20 for the candidate with the least number of alternatives (only if there is more than one candidate to assign a binding to a variable), a weight of 1 for every constraint that is not solvable and a weight of 10 for every binding in another branch. These weights were obtained by running ParAKL with four different values for each of the weights: more than 4000 runs were created each with a different weight calculation.
The weights we choose, actually gave the lowest number of nondeterministic rule applications for all benchmarks, except for Zebra, whose minimum was 255 for a different set of weights. So it seems that the chosen weights are near to optimal for a range of test programs and that a general selection algorithm is practical.
The following results can be found (using the general weight calculation, i.e. without forward checking):

Benchmark	ParAKL	SICS AKL
Money_1	84	127
Money_2	84	76284
Queens(8)_1	18	23
Queens(8)_2	146	336
Zebra	258	540

Table 1: Number of nondeterministic rule applications

Money_1 and money_2 are the same program using member/2 (to assign values to the variables) and difflist/1 (to make sure that all the values assigned to the variables are unequal to each other), but money_1 reorders the goals to make sure that carry/1 for the leftmost carry is the leftmost goal in the clause and, thus, will be selected first using the left-to-right strategy used by SICS AKL.
Queens_1 is the famous queens program but creates constraints for the rows, columns and diagonals to make sure that maximum one field is occupied. The second queens program is the usual version using the delete/3 predicate. Zebra is the usual zebra program in which difflist/1 creates the inequality constraints between the different program variables.
SICS AKL is a sequential AKL implementation by SICS, which uses left-most selection for nondeterministic promotion.

1. A weight of 5 is assigned if the constraint is an inequality constraint.

The results show that a big improvement can be made with a simple selection criterium. It is also clear that the selection algorithm stabilizes the execution: the reordering of goals in the Send-More-Money puzzle has no effect on the number of nondeterministic promotion steps.

4. Forward checking

The selection algorithm, described in the previous section, can be extended by adding some sort of forward checking *after* the selection algorithm is applied, but *before* the copying algorithm. Using this forward checking some alternatives of the selected candidate goal can be removed, because a nondeterministic promotion of one of the alternatives would definitely lead to a failure. For example:

```
a(1).
a(2).

b(2).
b(3).

:- a(X), b(X).
```

If the first alternative of a/1 is promoted nondeterministically, the corresponding or-branch would fail, leaving one alternative for a/1 which is promoted deterministically. However, if forward checking is performed on the first alternative of a/1, it would remove that alternative without applying the (rather expensive) nondeterministic promotion rule, because there is no corresponding binding for the variable 'X' in b/1. So, instead of doing a nondeterministic promotion, a forward check is performed making a/1 deterministic.

4.1. The forward checking algorithm

Forward checking algorithms are very well known in the area of solving constraint satisfaction problems [4][7]. In this area forward checking algorithms are used to remove values from the domain of a variable. This is similar to the way our forward checking algorithm works: instead of removing values from a domain, alternatives from a candidate goal are removed. The similarity is clear, because the domain of a variable can be represented by a goal with as many clauses as there are domain values. For example, the domain of variable 'X', which is equal to [1..4], can be represented as follows:

```
domain_X(1).    domain_X(2).    domain_X(3).    domain_X(4).
```

It is clear, however, that our algorithm is different from an ordinary forward checking algorithm: our algorithm has to check for unification consistency. The different structure of the computation tree (for AKL) alters the basic algorithm as well.

The forward checking algorithm is defined as follows:

```
ForEach binding made in the alternative to an external variable DO
    IF binding inconsistent with surrounding and-box THEN
        Remove alternative from candidate
    FI
OD
```

Checking of the consistency of a binding with its surrounding and-box is testing whether all the suspensions of the variable are consistent. This means, if a suspension is from the surrounding and-box and is inconsistent, then the binding is inconsistent. On the other hand, if a suspension is from an alternative whose choice-box is directly allocated under the surrounding and-box and all alternatives from this choice-box are inconsistent with the binding, then the binding is inconsistent[1].

If an extra binding is created (because of the successful application of a constraint, for instance), this binding is checked as well. This was very useful, especially for programs with complex constraints (like, for instance, the Send-More-Money puzzle).

4.2. The implementation results

The algorithm was implemented in ParAKL and the same programs as in the previous section were run. The results are summarized in table 2: the ParAKL and SICS AKL columns have not changed. The reason why the inclusion of the forward check does not lower the number of nondeterminisitc rule applications for 'Queens(8)_1' and 'Zebra', is not clear at the moment.

Benchmark	ParAKL + forward checking	ParAKL	SICS AKL
Money_1	49	84	127
Money_2	49	84	76284
Queens(8)_1	18	18	23
Queens(8)_2	52	146	336
Zebra	258	258	540

Table 2: Number of nondeterministic rule applications

1. This was shown in the example.

4.3. Overhead of the algorithms

For the two versions of ParAKL (with and without forward checking) the relative overhead compared to the complete execution time for every part of the nondeterministic promotion step was investigated as well (using the queens_2 program):

Algorithm	ParAKL + forward checking	ParAKL
Selection algorithm	18.5%	10.0%
Forward checking	14.5%	-
Copying algorithm	23.4%	50.4%
Total overhead:	56.4%	60.4%

Table 3: Relative overheads compared to total execution

These results show that the amount of overhead is limited, especially if the forward checking algorithm is included. The reason for the higher overhead for the selection algorithm in the left column is that the number of selections is greater than the number of copying steps because the forward checking algorithm removes some selected alternatives before they are copied: the result is a 4-fold speedup in absolute time for this benchmark, while ParAKL is about the same speed as SICS AKL. We have also tried to run the program with the simplest selection criterion possible: always select the first goal in the list of available goals. With this criterion, the program could not even run successfully.

5. Conclusions

There are two observations that can be made: first, an intelligent selection algorithm improves the execution of search programs in AKL by a factor of 2 to 3 in terms of the number of nondeterministic rule applications. We believe that this improvement can be obtained, generally speaking, in the Andorra model as well.
Second, the use of these techniques stabilizes the execution time of search programs, i.e. it becomes more independent of the order of the goals in a conjunction (this was shown in the two Send-More-Money programs).
If one looks at the overhead of the two described algorithms, it is obvious that the price that is payed, due to the extra computation of the priorities, is compensated by the smaller number of copying steps.

There is a disadvantage to the described techniques: the programmer loses the ability to direct the search by changing the ordering of goals in a conjunction. This is an advantage as well if one looks at it from a different angle: the programmer is freed from the necessity to have knowledge of the underlying execution model. This is especially important if one programs in a parallel execution environment.

6. Future work

As ParAKL is a parallel execution model for AKL it is obvious that the described techniques must be parallellised as well. This is one of our major tasks that has to be performed to have a fully functional, parallel implementation for AKL.
The described techniques can be further optimized and evaluated. Especially the weights, that are used in the selection algorithm, can be fine-tuned to satisfy a large class of search programs.
Both the algorithms must be further optimized (as well as the copying algorithm), because the overhead for the algorithms will increase when the number of copying steps decreases.
One of the ideas for improvement is to maintain some sort of treshold for the priority calculation: if an inspected candidate goal has a calculated priority that is greater than the treshold, select this goal. This treshold can be used as a memory cell as well: the priority calculated for the selected candidate goal is saved in the treshold. This means that, if a candidate goal is found, in the next nondeterministic promotion step, which has a priority greater than the priority of the currently selected goal, select this goal; its priority is large enough (but probably not the largest). This idea could decrease the number of inspected candidate goals.
Finally, there are possibilities for reducing the overhead of the calculation of the priorities: global analysis can deduce freeness of variables and whether they are constrained or not, so that not all arguments of a goal must be inspected during the calculation of the first priority. And the forward checking priority calculation can be sped up by generating a form of interprocedural indexing code.

7. Acknowledgments

The authors wish to thank the ‘Diensten voor de Programmatie van Wetenschapsbeleid’ of the Belgium Ministry for support through project IT/4 and the European Community for support through Esprit Project ACCLAIM (07195).

8. References.

[1] S. Abreu, L.M. Pereira and P. Codognet, “Improving backward execution in the Andorra Family of Languages”, in *Proceedings of the Joint International Conference and Symposium on Logic Programming*, ed. Krzysztof Apt, MIT Press, Cambridge, Massachusetts, 1992.

[2] Sverker Janson, Seif Haridi, “Programming Paradigms of the Andorra Kernel Language”, in *Proceedings of the International Logic Programming Symposium*, ed. V. Saraswat, The MIT Press, Cambridge, Massachusetts, 1991.

[3] Sverker Janson and Johan Montelius, “Design of a sequential prototype implementation of the Andorra Kernel Language”, Technical Report, SICS, september 1991.

[4] Alan K. Mackworth, “Consistency in Networks of Relations”, Artificial Intelligence, Volume 8, pp.

99-118, 1977.

[5] A. Marien, B. Demoen, "Findall without findall/3", in *Proceedings of the Tenth International Conference on Logic Programming*, ed. David S. Warren, MIT Press, Cambridge, Massachusetts, 1993.

[6] Remco Moolenaar and Bart Demoen, "A parallel implementation for AKL", in *Programming Language Implementation and Logic Programming: PLILP '93*, Tallinn, Estonia, 1993.

[7] Bernard A. Nadel, "Constraint satisfaction algorithms", Computational Intelligence, Volume 5, pp. 188-224, 1989.

[8] Vitor Santos Costa, David H.D. Warren, and Rong Yang, "Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-Parallelism", in *Proceedings of the Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming, PPOPP'91*, SIGPLAN NOTICES, vol. 26, no. 7, pp. 83-93, ACM Press, Baltimore, July 1991.

[9] Jeffrey D. Ullman, "Principles of Database Systems", Computer Science Press, Rockville, 1982.

[10] Van Hentenryck, "Constraint Satisfaction in Logic Programming", The MIT Press, Cambridge, Massachusetts, 1989.

[11] Warren, David H.D., "The Andorra principle", Internal report, Gigalips Group, 1988.

A Shared-Memory Parallel Execution Scheme of KLIC

Masao Morita, Nobuyuki Ichiyoshi
Mitsubishi Research Institute [*]
morita@boss.mri.co.jp, ichiyoshi@mri.co.jp

Takashi Chikayama
Institute for New Generation Computer Technology (ICOT) [†]
chikayama@icot.or.jp

March 1994

Abstract

An extension of the KLIC implementation of KL1 for shared-memory multiprocessors is being designed and tested. The design goal was to add shared-memory parallel processing features without modifying the sequential core implementation, by using generic objects. Apart from the obvious merit from the software engineering standpoint, the separation minimizes the overhead imposed on the sequential core by a parallel extension. It is also justified by the growing discrepancy between local memory access cost and shared memory access in multiprocessors using high-performance processors. A new garbage collection (GC) scheme, in which each processor can independently garbage-collect the shared memory area while others are working, is also introduced.

1 Introduction

KLIC [2] is a portable implementation of parallel logic language KL1 [12, 1]. It is written in C, and compiles a KL1 program into a C program. KLIC is intended as a vehicle for porting the software that was developed in the Japanese Fifth Generation Project which would otherwise runs efficiently only on the specially-built parallel inference machines (PIMs) [11]. More importantly, it is to be a platform for application programmers — anywhere in the world — to write parallel programs in KL1 for standard hardware. KLIC has a framework called "generic objects" for introducing new features, such as linking with foreign language subroutines, and with numerical or graphics libraries. To introduce a new class of generic objects, the user is to provide standard methods, such as deref(), unify(), gc() and print(). (These method are called implicitly from within the kernel code or from builtin predicates.) The user can also provide methods specific to the class. (A special syntax is used to call them: `generic:`*Method*(*Args*).)

A sequential implementation runs on various workstations and proves to be reasonably efficient (approx. 2 MLIPS peak performance on SPARCstation 10 with 36 MHz SuperSPARC processor). Currently, work is going on to develop parallel versions for shared-memory multiprocessors and distributed-memory parallel computers. This paper

[*]2-3-6 Otemachi, Chiyoda-ku, Tokyo 100, JAPAN
[†]1-4-28 Mita, Minato-ku, Tokyo 108, JAPAN

describes the design rationale and implementation outline of the shared-memory parallel version of KLIC. Distributed-memory parallel version of KLIC is described in [9].

Shared-memory implementations of concurrent logic languages developed so far have taken either a UMA model (all memory is shared) or a NORMA model (no memory is shared). Examples of the UMA scheme include the shared-memory execution scheme in the PIM implementations [7] of KL1 and the JAM [3] implementation of Parlog [6]. Except for the local goal stack and various control data per processor, all data in the heap area are shared. Since a variable may be simultaneously accessed by multiple processors, its instantiation requires locking: lock the variable, check if it is still uninstantiated, write on it, unlock it. The problem is that a variable is always locked even if it is accessible by only one processor, thus degrading the overall performance. The implementation of Strand [5] for shared-memory multiprocessor is an example of the NORMA scheme [4]. Each processor has a private memory area, and inter-processor communication is realized by writing and reading communication buffers (one for each processor). In effect, it is a shared-memory porting of a distributed-memory implementation. A merit of this scheme is that sequential execution speed does not suffer from the parallel extension. (Locking is necessary only when writing to a message buffer.) However, the software overhead in interprocessor communication is considerably larger than the base hardware overhead (shared-memory read/write).

The design rationale of the shared-memory implementation of KLIC is to avoid the defects in both of the above schemes. It tries to minimize the new overheads in the sequential execution by separating the local and shared area, while keeping interprocessor communication overheads reasonably low by directly reading and writing shared area. Under this scheme, the garbage collection of the local heaps can be done independently. The garbage collection of the shared area is also designed so that as little synchronization between processors as possible is necessary.

2 Outline of the Shared-Memory Parallel Execution Scheme

2.1 Design Goals

Because a high priority of the design is to not degrade the sequential performance of KLIC, it is decided that the shared-memory parallel version of KLIC should not have a special kernel which differs largely from the sequential base kernel (which is intended to be more or less optimal for sequential execution), but it should be realized by introducing the shared-memory parallel processing feature by using generic objects. This is in keeping with the policy in KLIC that add-on features be introduced by defining suitable generic objects. Apart from the obvious merits from the software engineering standpoints, it prevents the added features (expected to be invoked not too often) impose overhead on the sequential core code (which mostly determines the performance).

The most important difference between operations on KL1 data in sequential implementation and shared-memory implementation is that there are shared variables in the latter. (By definition, a *shared* variable is one which may be accessible by more than one processor.) A shared variable needs a special treatment, such as locking is necessary for instantiating it. Therefore, shared variables are introduced as generic objects. Non-shared variables remain a kernel data type. Only those variables guaranteed to be not shared by multiple processors can be non-shared variables. In order to actually differentiate between shared and non-shared variables, the memory area is divided into local areas and one shared area (Fig. 1). Each processor owns a local area which only it can read and write on. There can be pointers from a local area to the shared area, but the opposite is not allowed. Thus, a variable in a local area (called *local variables*) are guaranteed not accessible by processors other than the owner of the local heap, and a variable in a shared area (called *shared variables*) are (potentially) shared. A local variable is the same as a variable in sequential KLIC (represented by a self-reference), while a shared variable is a generic object of class SHVAR. By default, a processor allocates goals and data in the local area.

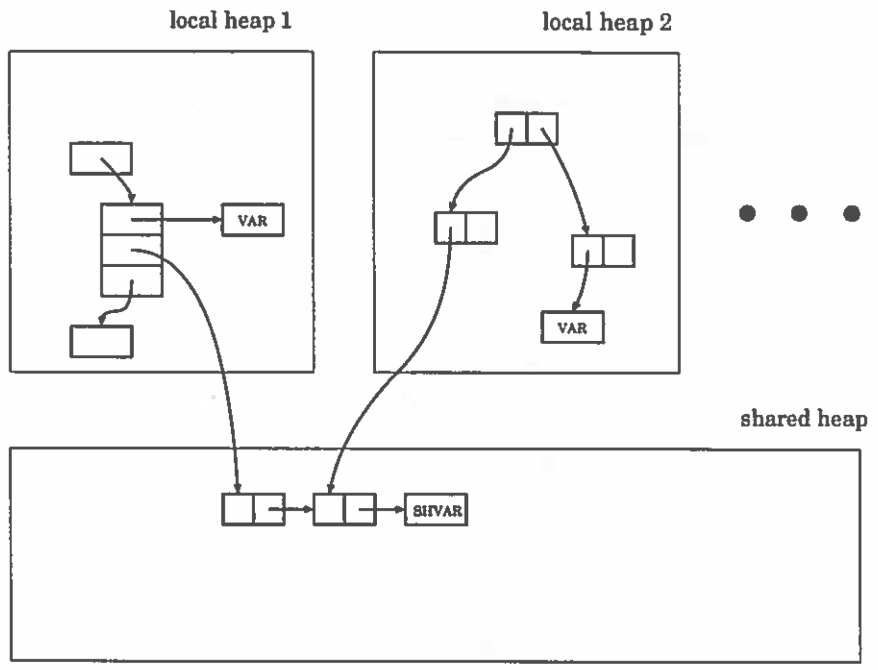


Figure 1: Local and Shared Heaps

2.2 Local and Shared Areas

The memory is divided into local areas and a shared area. Each processor has a local area which it can read and write without coordination with other processors. A processor can also read and write on the shared area, but it must do so with coordination with other processors since some of them may be accessing the same data simultaneously.

A local area consists of the maintenance data for local execution and local heap for dynamic allocation of local goals and KL1 data. The maintenance data includes the local heap top pointer, the root of the local goal stack, etc. The shared area consists of the maintenance data for processor interaction and shared heap for dynamic allocation of shared goals and KL1 data. The shared maintenance data includes external goal pools (one for each processor), an interrupt flag for each processor, etc. An external goal pool of a processor holds the goals given to the processor by other processors for load distribution. Thus, goal are inserted by processors other than the owner processor and removed by the owner processor for execution. The interrupt flag is set to alert the owner processor at the slit check timing [2]. It is the same as the one in the sequential KLIC, but is now allocated in the shared area, so that not only the owner processor, but also other processors can set it.

Each local heap area is divided into two spaces for copying garbage collection. The shared heap area is divided into three spaces that are used in a circular manner. At any instant, there is one old space, one new space, and one unused space. Active data are in the old or the new space, or it may extend over both of them, but there must be no pointers from the new space to the old space.

Each processor has one "current" shared heap space (either the old or the new) for allocation of shared data. Each space in turn has a processor table listing those processor whose current space is the space. Each processor keeps a page of shared heap for allocation of shared goals and data. When the page has run out, the processor demands a new page from the current shared heap space. (In the current implementation, the size of one page is 4KB.) A processor "moves" from the old space to the new space when it performs copying garbage collection of the old space. The absence of pointers from the new space to the old guarantees that all data in the old space has become garbage when there the last processor moves to the new space (i.e., the new space becomes its current shared heap space). Thus, the old space can become an unused space at that point.

2.3 Goal Distribution

A goal spawned by a processor can move to another processor, only when explicitly specified by a goal distribution pragma.[1] When a goal moves, the data directly and indirectly referenced by it are copied to the shared area.

For example, in the following code,

```
p(X) :- ... | ..., q(X,foo(Y))@node(4), ... .
```

the goal `q(X,foo(Y))@node(4)` is allocated in the shared area. It involves the copying of the data structure pointed to by variable `X` from the local to the shared area, and the allocation of the compound term `foo(Y)` in the shared area. `Y` is a new shared variable. If `X` is an uninstantiated variable, a shared variable is allocated in the shared heap and the local variable will be converted to a reference to the shared variable. After the allocation and copying, the goal is put in the external goal pool of processor 4. After the goal has been copied to the shared heap, the pointer to it is inserted into the external goal pool of the target processor. The current priority level of the target processor is then checked. When the priority of the newly created goal is higher than the current priority level of the target processor, the interrupt flag for the target processor is set so that the new goal may be scheduled right after the current reduction. Otherwise, the target processor is not interrupted. A processor checks its external goal pool whenever it moves to a lower priority level.

3 Garbage Collection

3.1 Garbage Collection of Local Heap

The separation of local and shared heaps allows each processor to perform local garbage collection independently. The sequential KLIC implementation has a copying garbage collector, and only small extension of it are needed. They are as follows:

- At the beginning of the local garbage collection, the remaining size of the current shared heap of the processor is checked. If it is below a certain threshold value, the processor starts garbage collection of the shared heap.
- The local garbage collector treats pointers to shared heap as leaf data (i.e., stops traversing the data structure that lies beyond).

[1] In contrast, load distribution is implicit in the shared-memory execution of the PIM implementations. An idle processor steals a goal from other processors' goal stacks.

3.2 Garbage Collection of Shared Heap

3.2.1 Asynchronous Garbage Collection

In most previous shared-memory parallel implementations, garbage collection of the shared heap is synchronous, i.e., all processors stops normal execution and garbage collection is started (possibly in parallel). However, as single processor performance increases much more rapidly than that of bus cycle speed, and since there is little locality in garbage collection, it is expected that the shared bus will become a bottleneck in parallel garbage collection in future multiprocessors. Thus, an asynchronous garbage collection scheme is designed for the shared-memory KLIC implementation, so that one processor can perform shared-memory garbage collection while others are executing normal code. For example, in Fig. 2 illustrates the situation in which there are two processors (PE1 and PE2). In this example, PE1 detects the shortage of the old space first (Fig. 2 (a)), and copies the active data it references to the new space (Fig. 2 (b)), while PE2 is still executing normal KL1 code. PE1 then resumes normal KL1 execution. Later, PE2 also detects the shortage of the old space and copies the active data it references to the new space (Fig. 2 (c)). The *old* old space (space A) now holds no active data, and it becomes the unused space, the *old* new space (space B) becomes the *new* old space, while the *old* unused space (space C) now becomes the *new* new space.

In case, PE1 has used up the new space before PE2 detects the shortage of the old space (Fig. 3 (a)), PE2 is interrupted and forced to do garbage collection. PE2 copies the active data in the old space to the unused space (Fig. 3 (b)). At this point, the *old* old space (space A) becomes the unused space, the *old* new space (space B) becomes the old space, and the *old* unused space (space C) becomes the new space. Since the *old* new space has run out, PE1 copies the active data in it to the *new* new space. Finally, *old* new space (space B) becomes the unused space, the *old* unused space (space C) becomes the old space. the *old* old space (space A) becomes the new space. (Fig. 3 (c)).[2]

3.2.2 Steps of Garbage Collection

A processor conducts a garbage collection of shared heap in the following steps, if its current shared heap space is the old space.

1. It first performs local garbage collection.

   The difference from a normal local garbage collection is that whenever a reference to data in the old shared heap space is encountered, it is memorized.

2. After the local garbage collection has finished, active data in the old shared heap space are copied to the new shared heap space.

   The roots of copying are those references which are memorized as above. How data in the shared heap are copied is described in the next subsection.

3. When the copying is over, the processor changes its current space from the old to the new space.

   It also removes itself from the processor table of the old space, and add itself to the processor table of the new space.

If the current shared heap space is the new space when the processor detects shared heap shortage, the processor sets the interrupt flags of the processor in the processor

[2]Since all processors must stop normal execution during a forced garbage collection, a stop-and-copy type parallel garbage collection scheme (e.g., [8]) could be employed.

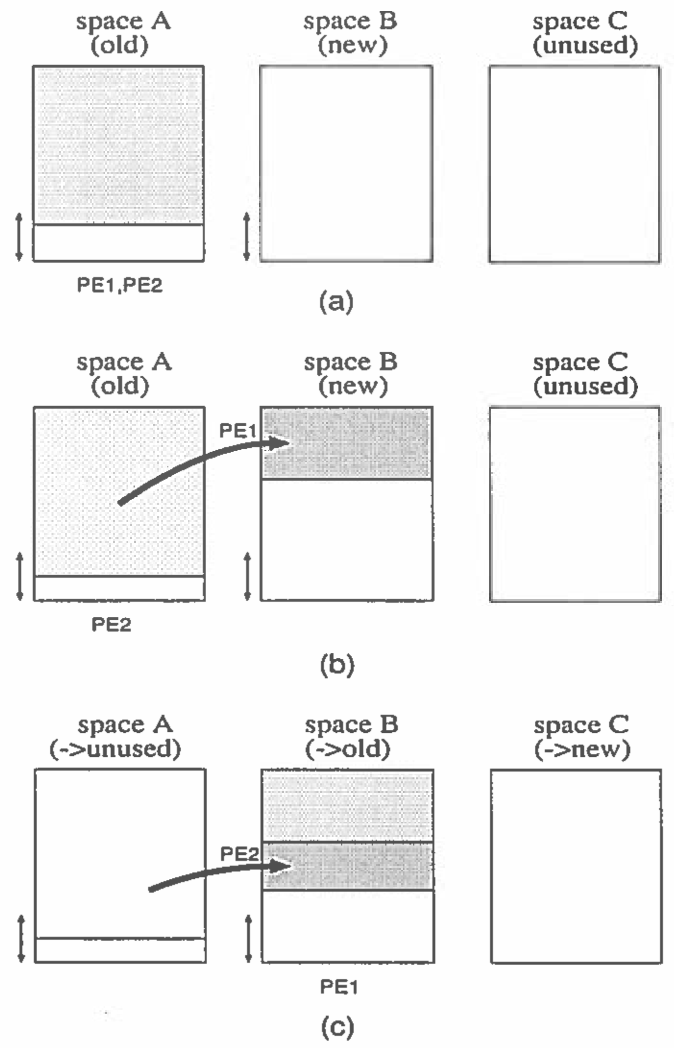


Figure 2: Asynchronous GC of Shared Heap

table of the old space to force garbage collection, as described in the previous subsection. After those processors have finished shared-heap garbage collection, the *old* new space will become the *new* old space, and the *old* unused space will become the *new* new space. the requesting processor then starts shared-heap copying garbage collection from the *new* old space to the *new* new space. In case, the *new* new space overflows, the system can choose to abort execution because it means the aomount of active data is more than can be accommodated in a single space. Alternatively, the *new* unused space could be used as the last resort. If all processors working the old space have moved to the new space before the unused space runs out, the system can continue execution.

Current implementation allows only one processor to perform shared-memory garbage collection at one time, in order not to complicate the copying algorithm.

3.2.3 Bottom-up Copying

Care must be taken in copying during shared-memory garbage collection, since processors other than the one doing garbage collection are executing normal KL1 code, accessing the same space. Ordinary breadth-first copying algorithm does not work, since it introduces

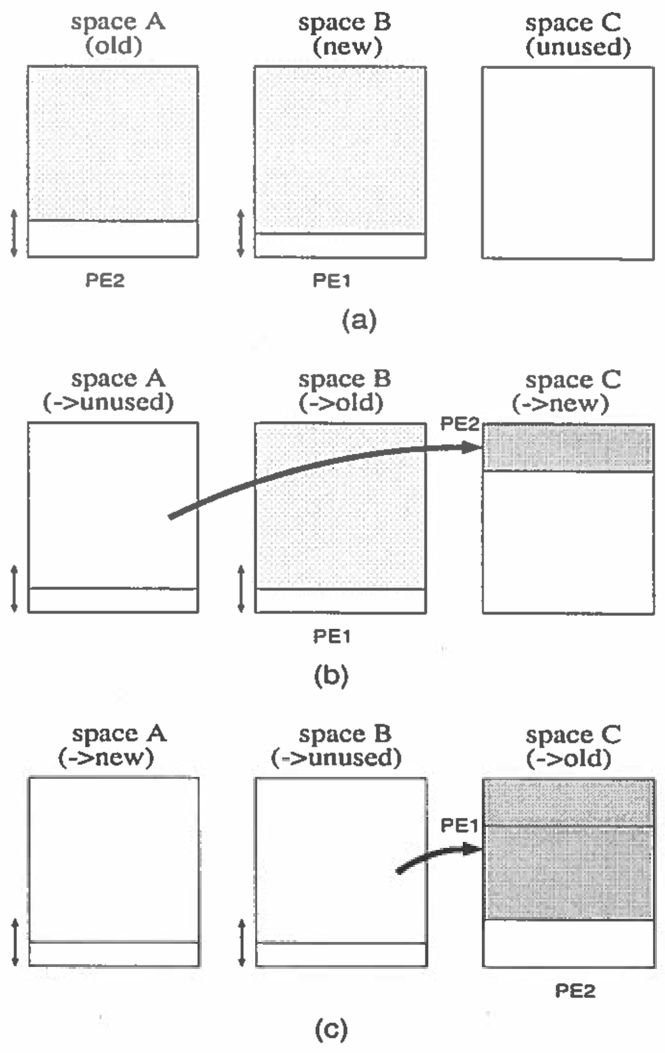


Figure 3: Forced GC of Shared Heap

illegal (uninitialized) KL1 data during garbage collection. Locking of data that are being copied is undesirable, because that would complicate the sequential core and degrade overall performance. A bottom-up copying described below is introduced to avoid this.

In the bottom-up copying scheme, data in the old space are copied bottom-up. Pointers to copied data in the old space are replaced by pointers to the corresponding copies in the new space, so that the copied data in the old space would become unreferenced.

For example, a copying of a reference (from a register or from a local data) to a cons cell (Fig. 4 (a)) is done as follows:

1. The data structure referenced by the car part is (recursively) copied (Fig. 4 (b)).

2. The old car part is replaced by the reference to the copied data structure in the new space.

3. The data structure referenced by the cdr part is (recursively) copied.

4. The old cdr part is updated to the reference to the copied data structure in the new space.

5. A cons cell is allocated in the new space and the contents of the old cons cell are copied.

6. The original reference to the old cons cell is updated by the reference to the cons cell in the new space (Fig. 4 (c)).

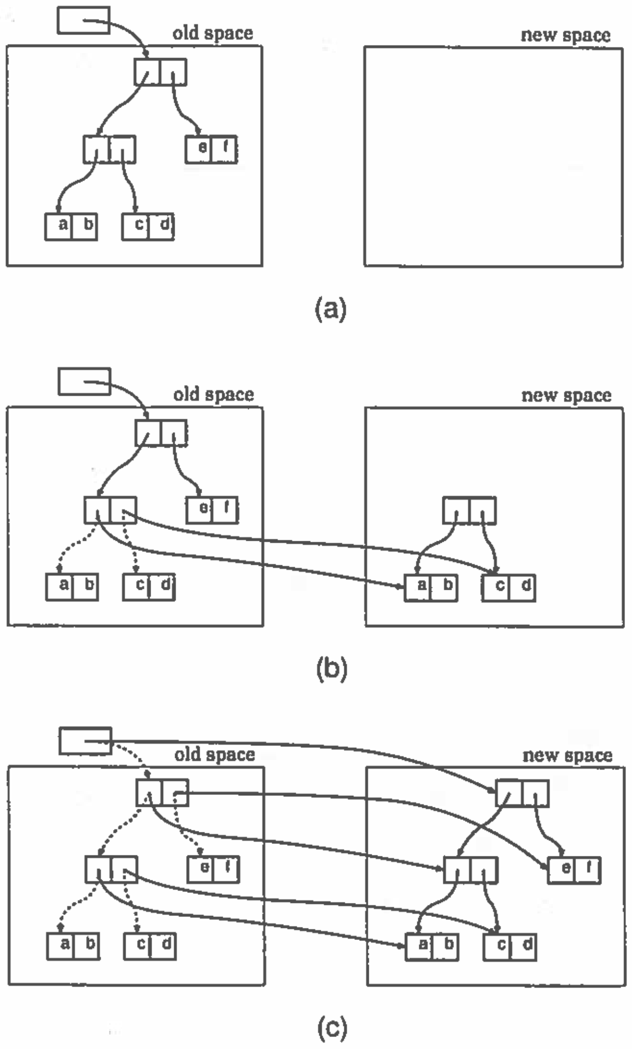


Figure 4: Bottom-up Copying

Note that at any time during the above procedure, other processors might reference the cons cell and/or the data structures directly or indirectly referenced by the cell, but that they see the same logical data whether the data lie in the old space or they lie in the new space. Since replacement of the new reference for the old is an atomic operation (simple store), no locking of data is necessary.[3]

In bottom-up copying, a special care must be taken to handle circular data structure correctly. The correct algorithm is under development.

[3]However, the store order observed by the processors running the normal code must be the same as that issued by the copying processor. Recent multiprocessor workstation requires a store barrier instruction to guarantee this (cf. [10]).

4 Current Status

A test implementation of the shared-memory parallel execution scheme of KLIC is being developed on a multiprocessor SPARCstation running Solaris 2.1. The code for normal execution of KL1 is mostly finished and initial performance measurements have begun. Shared-memory garbage collection algorithm is still being debugged and refined. Future work includes investigation of ways to reduce the overhead involved in data copy from local to shared heap, investigation of parallel shared-memory garbage collection, and development of a distributed shared-memorires implementation scheme (e.g., for network-connected multiprocessors).

Acknowledgments

We thank the members of the KLIC development group and the KLIC working group for valuable discussions.

References

[1] T. Chikayama. Operating system PIMOS and kernel language KL1. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1992*, pages 73–88, 1992.

[2] T. Chikayama, T. Fujise, and D. Sekita. A portable and efficient implementation of KL1. In *ICOT/NSF Workshop on Parallel Logic Programming and Its Programming Environments*, 1994.

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

[4] I. Foster. *Systems Programming in Parallel Logic Languages*, page 89. Prentice Hall, 1990.

[5] I. Foster and S. Taylor. *Strand — New Concepts in Parallel Programming.* Prentice Hall, 1990.

[6] S. Gregory. *Parallel Logic Programming in PARLOG — The Language and its Implementation.* Addison-Wesley, 1987.

[7] K. Hirata, R. Yamamoto, A. Imai, H. Kawai, K. Hirano, T. Takagi, K. Taki, A. Nakase, and K. Rokusawa. Parallel and distributed implementation of concurrent logic programming language KL1. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1992*, pages 436–459, 1992.

[8] A. Imai and E. Tick. Evaluation of parallel copying garbage collection on a shared-memory multiprocessor. *IEEE Transactions on Parallel and Distributed Systems*, 4(9), 1993.

[9] K. Rokusawa, A. Nakase, and T. Chikayama. Distributed memory implementation of KLIC. In *ICOT/NSF Workshop on Parallel Logic Programming and Its Programming Environments*, 1994.

[10] SPARC International. *The SPARC Architecture Manual, Version 8.* Prentice Hall, 1992.

[11] K. Taki. Parallel inference machine pim. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1992*, pages 50–72, 1992.

[12] K. Ueda and T. Chikayama. Design of the kernel language for the parallel inference machine. *The Computer Journal*, 33(6):494–500, 1990.

The DEC CRL compiler for pH (a parallel Haskell)

Rishiyur S. Nikhil
Digital Equipment Corporation, Cambridge Research Laboratory
One Kendall Square, Building 700, Cambridge MA 02139, USA
Email: `nikhil@crl.dec.com`

Extended Abstract, ***DRAFT*** of March 2, 1994

1 Introduction

We are nearing completion of a first implementation and release of pH, a parallel dialect of Haskell[6] that has strong roots in the parallel programming language Id[10]. Our primary target is workstation farms, *i.e.*, standard workstations connected with standard networks. This paper provides an overview of our approach and compilation system.

2 pH, a parallel dialect of Haskell

pH is a parallel dialect of Haskell. Figure 1 illustrates where pH fits into the family of functional programming languages. pH syntax is a proper superset of Haskell syntax[6], and uses the same Hindley-Milner static polymorphic type system. It borrows its dynamic semantics ("lenient"), and some language extensions, from its predecessor, Id[8]. The language extensions are mainly two kinds of non-functional data structures. I-structures are write-once data structures with implicit synchronization, and are closely related to logic variables in logic programming languages. I-structures sacrifice referential transparency, but preserve determinacy. M-structures allow multiple updates (and introduce non-determinism), but also have implicit synchronization of reads and writes. A more complete discussion of these language features may be found in [10].

3 The target: C code on workstation farms

Our main target is workstation farms (or clusters), which are collections of standard workstations connected by standard networks. The main criticism of these platforms for parallel processing is their poor communication performance, compared to custom parallel machines (such as the Thinking Machines CM-5, Intel Paragon and Meiko CS-2). However, the communication performance gap is shrinking rapidly, and we believe that it will soon be insignificant.

For example, our current target is a collection of Digital Alpha workstations running the OSF/1 Unix operating system, connected by a Digital Gigaswitch, which is a crossbar switch interconnecting upto 22 FDDI rings. This platform can achieve full FDDI communication bandwidth (12.5

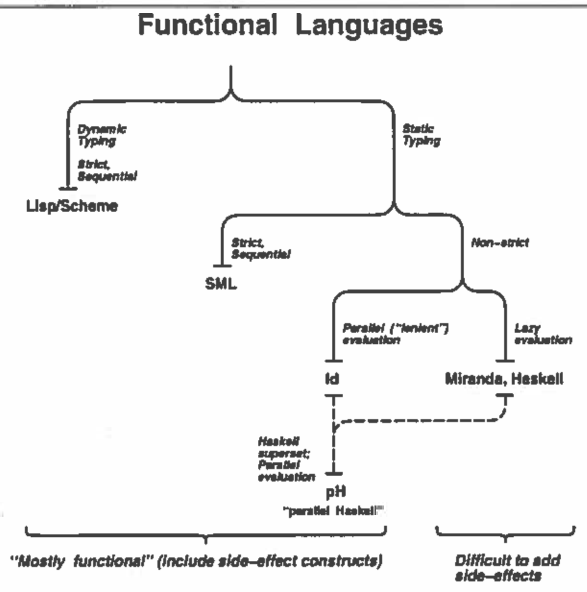


Figure 1: pH and other functional programming languages

MBytes/sec) simultaneously between all pairs of processors. This number is not far below what is achievable in custom parallel machines, and when we move to ATM interconnects (likely within a year), the bandwidth gap will close completely.

On the latency side, while this platform has excellent performance by modern networking standards (just over 300 microsecs, user-to-user, for a 1-word UDP/IP message), custom parallel machines typically communicate a small message in a few tens of microseconds. We believe that workstation farm message latency can drop below 100 microseconds on current hardware, with better communication software,[1] and that the gap will close completely when we move to ATM networks and better network interfaces.

With these trends, we believe that workstation farms offer the best potential vehicle for widespread use of parallel processing.

Like many other researchers, we have chosen to produce C code for portability. Experience with the Glasgow Haskell compiler indicates that the limitations of C as a "portable assembler" can be overcome with just a few architecture-specific post-processing tweaks to compiler output [12].

4 Dealing with latency, a central issue

Whether the target platform is a workstation farm or a custom parallel machine like the CM-5, communication latency is a central issue, and is likely to remain so because processor speeds are increasing much faster than communication speeds.

In general-purpose parallel languages like pH (and CLPs), latency also arises out of unpredictable synchronization waits. Unlike HPF [4], where the compiler carefully orchestrates computation and

[1] We assume the farm is on a local area network where we can supply the fast communication software to all components.

communication schedules, a compiler for pH cannot predict how long a consumer of a datum will have to wait for the producer to make it available.

We use (and plan to use) several techniques to deal with latency. First, we expect to use object-level cacheing: when a thread on processor P1 tries to access an object that was allocated on processor P2, the object is copied over to P1; subsequent accesses from P1 access that local copy. This is related to work on distributed shared memory and directory-based cache-coherence [1, 5]; the main difference is that we cache objects, not pages or cache-lines, and we rely on language-level support to avoid maintaining directory information wherever possible. We expect that this should substantially reduce the overheads of cache coherence.

Second, we use multithreading to avoid idling while waiting for a global object access or waiting for a synchronization event. Again, we rely on language-level support to know when to switch threads. Multithreading is lightweight– we do it entirely in the runtime system (no OS involvement), without pre-emption, *i.e.,* it is completely event-driven.

Finally, a potentially remote access may, in fact, be local because the accessor and the accessee turn out to be allocated on the same processor. To exploit this locality, we have various optimizations that allow communication through registers instead of messages (these are described in [9]).

5 Overall structure of the compilation system

Figure 2 shows the overall structure of our compilation system. The front end is being constructed

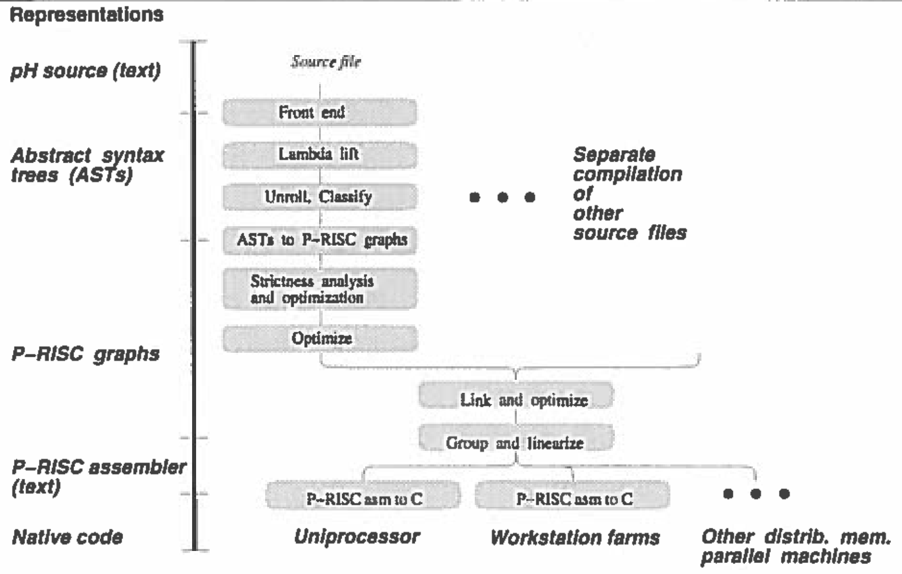


Figure 2: Overall structure of the DEC CRL compilation system for pH.

by modifying the Glasgow Haskell compiler [12] and includes parsing, typechecking, overloading resolution and specialization, and desugaring of pattern-matching. The lambda-lifter is quite conventional [7].

The Unroll-and-Classify phase handles currying, in two stages. First, it "unrolls" each curried function of arity n into a series of $n+2$ definitions. The first definition simply binds the original function identifier to a closure with zero partially-applied arguments. The next n definitions represent intermediate partial applications— the j'th function takes two arguments– an explicit environment containing $j-1$ partially applied arguments and a new argument, and produces a closure of the $j+1$'st function and an environment that includes this new argument. The last-but-one definition, which represents the final argument application, unpacks arguments from the environment and performs an ordinary, first-order call to the last function, which is an ordinary uncurried function of all the arguments. Second, the Unroll-and-Classify phase classifies and transforms every call site into one of three forms. If the function is unknown (so, it is a closure value), it produces an application of the code of the closure to two arguments, the environment of the closure and the new argument. If the function is a known function of arity n and applied to fewer than n arguments, this is replaced by a closure-building operation. Finally, if the function is a known function of arity n and applied to n arguments, we replace it by a direct call to its uncurried version.

At this point the compiler changes the program representation, from abstract syntax trees to P-RISC graphs. Since P-RISC graphs are perhaps the most unusual aspect of the compiler, we devote the next section to describing them.

6 P-RISC graphs

P-RISC graphs are a compiler intermediate language for the representation, analysis and optimization of fine-grain multithreaded code. P-RISC graphs include:

- Lightweight, compiler-specified microthreads, plus a model of locality in distributed memory machines;
- Split-phase, non-blocking, event-driven transactions to deal efficiently both with communication latency and synchronization waits, and
- Fine-grain synchronization at the level of individual data.

P-RISC graphs are parallel flowgraphs and are a proper superset of conventional flowgraphs. The fine-grain parallelism expressible in P-RISC graphs may also be used for instruction-level scheduling such as pipeline, superscalar, VLIW scheduling (although they have not yet been so used, to date). Although we use P-RISC graphs to represent pH programs, we believe that they are equally suitable for other declarative or imperative languages, such as parallel C, Fortran, CLPs, *etc.*

6.1 P-RISC parallel machine model

The P-RISC parallel machine model is shown in Figure 3. It consists of $N \geq 1$ nodes, each with its own processor and local memory. A local pointer (or *lptr*) is just a local address within a node. A global pointer (or *gptr*) is conceptually a *(node number, local address)* pair. *Instruction pointers* (or *IPs*), are traditional instruction addresses; they are always lptrs. Each stack resides entirely within a single local memory.

A *microthread* (or just *thread*, for short) is the smallest unit of parallelism. It is always associated with a particular stack, and is executed only by the processor that contains that stack. Of course, a microthread can initiate a new microthread in another stack, which may be on another node (details to follow). A node may execute more than one microthread simultaneously (if it is a multiprocessor),

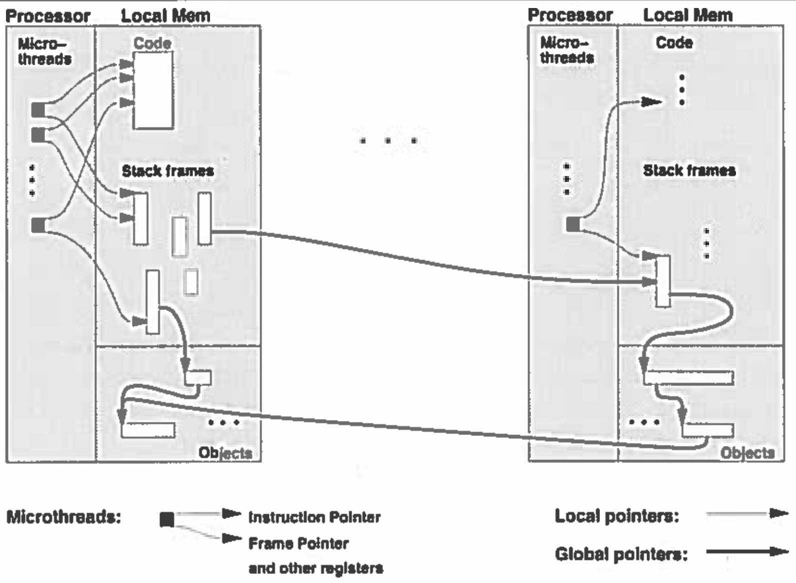


Figure 3: P-RISC parallel machine model

but most implementations execute only one at a time. There may be many microthreads per stack. This is in contrast to other popular models where a stack is usually associated with a single thread or process.

There is no microthread suspension or pre-emption. All operations that have unpredictable latencies (such as remote memory accesses or synchronization waits) are structured by the compiler as *split-phase operations*: a microthread T1 issues a request which contains a "continuation" microthread descriptor T2; when the operation completes, T2 is enabled (and may run in parallel with T1).

The state of a microthread is described by just two local pointers: an instruction pointer IP and a stack pointer FP. A microthread may have more registers, but since there is no notion of suspension, a microthread is responsible for its own register initialization on startup and register saving before it dies.

A ***codeblock pointer*** or ***cbptr*** is a machine-wide identifier for a function, and is used to invoke functions, possibly across processors. We separate this concept from the code address of a function since the same function may, in general, be loaded at different addresses in different nodes.

6.2 P-RISC primitive machine types

In addition to traditional primitive data types (integers, floating point, *etc.*), local pointers (lptrs), instruction pointers (IPs) and stack pointers (FPs) are just local memory addresses. We treat global pointers (gptrs) as primitive machine types, although they are conceptually structures containing a node number (integer) and an lptr (several operations need to extract this information from a gptr). We treat codeblock pointers (cbptrs) as primitive machine types, although they will usually be implemented as lptrs or integers.

For each primitive type T that can be contained in a heap object, there is also a corresponding *synchronized type* sT. In addition to containing a value of type T, an element of type sT also contains a *state* which has the values FULL and EMPTY. As we shall see, there are special remote memory access operations whose semantics depend on this state.

6.3 Program Structure

A program represented by P-RISC graphs consists of a collection of uniquely-named **Global** and **Codeblock** declarations which have the forms shown in Figure 4. A **Codeblock** declaration represents

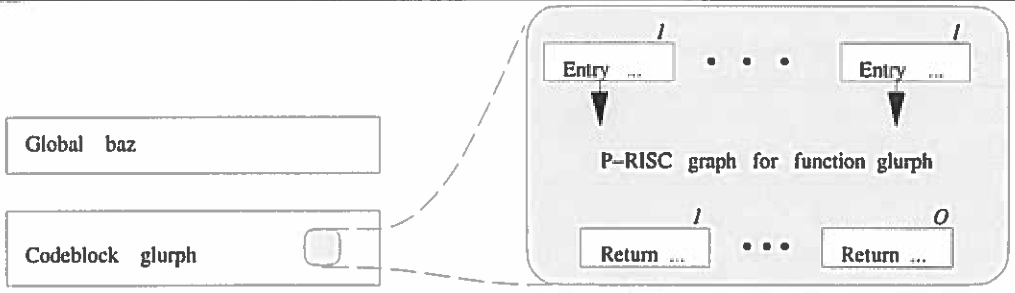


Figure 4: P-RISC graphs program structure

a function named **glurph** with $I \geq 1$ input threads and $O \geq 0$ output threads (we shall describe **Entry** and **Return** instructions shortly). The symbolic name **glurph**, when used in a thread, represents a cbptr. The codeblock declaration contains a P-RISC graph representing the function body.

The **Global** declaration represents a globally accessible location that is intially empty. Some thread in the program is responsible for giving it a value using a GIstore instruction, making it full. Other threads access the value using GIload instructions. These instructions are described in Section 6.7.

6.4 P-RISC Graphs: general features

A P-RISC graph represents read-only code. It is a control flow graph, *i.e.*, nodes in the graph are instructions, and edges represent control flow. However, P-RISC graphs are *parallel* control flow graphs, *i.e.*, there can be multiple simultaneous loci of control in the same graph. A locus of control is a microthread which, at a minimum, contains two registers: an instruction pointer IP (a point in the graph) and a frame pointer FP which points into a (local) stack. A microthread may contain more registers, but we leave this unspecified at the moment.

In general, an instruction has the form:

```
    d_t1, ..., d_tM = op_t [literals] (s_t1,...,s_tN)
```

i.e., it has destinations, an opcode, literals and sources (depending on the opcode, some of these may be missing).

Sources and destinations may be registers, or local memory locations (register indirect, perhaps with indexing). Here, for simplicity, all sources and destinations will be of the latter form, and in particular they will be FP-relative local memory locations. However, in our compiler, there is a stage where all sources and destinations are restricted to be registers.

The _t suffixes designate the type: for sources and destinations, it represents their interpretation, and for opcodes, it identifies a member of a related family. We use the following suffixes:

type suffix	*type*	*meaning*
`i`	int	integer
`ip`	lptr	instruction pointer
`fp`	lptr	frame pointer (pointer into a stack)
`nd`	int	node number
`gp`	gptr	heap address
`cb`	cbptr	codeblock pointer

6.5 Control Operations

The control operators are shown in Figure 5. **`Entry`** and **`Join`** are the only operators that begin a thread. **`Join`**, **`Halt`** and **`Prog_halt`** are the only operators that can end a thread. **`Join`** and **`Merge`** are the only P-RISC graph nodes that may have more than one input.

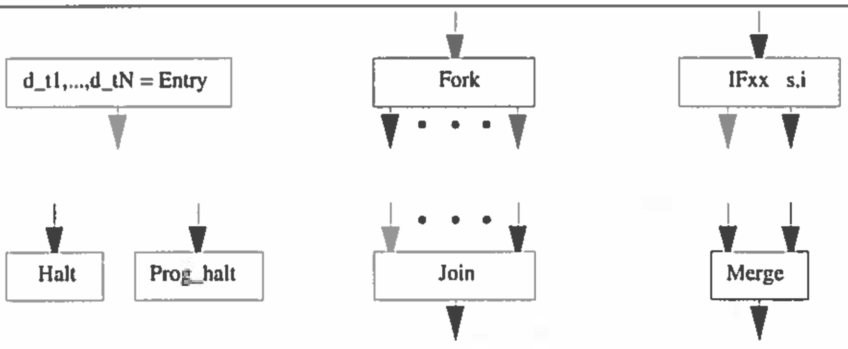


Figure 5: Control operations in P-RISC graphs.

`Entry` begins a thread. It is used at the top of a function body to receive arguments, below a function call to receive results, and below global memory operations to receive memory responses. In all cases, the arguments (function arguments or message payload) are received into the specified destinations ($N \geq 0$), and the thread continues along the output edge. It is the compiler's responsibility to ensure that the destination list matches the arriving arguments in number and type.

`Halt` ends the current thread. **`Prog_halt`** also asserts that the program has completed. **`Prog_halt`** does not do global termination detection— in a correctly compiled program, the thread that executes **`Prog_halt`** will be the only remaining thread executing.

`Fork` forks off one or more new threads (all with the same, current, FP) from the original thread, all of which may then run in parallel. Note, there is no resource allocation involved in a fork (unlike traditional notions of fork)– it is extremely lightweight. **`Join`** is a synchronizing operator. When all input threads have arrived, a single thread continues. Note that none of the input threads suspend– they simply die. **`Join`** is readily implemented with a simple counter in the current frame, initialized to zero. Each incoming thread (atomically) increments the counter and compares it to n, the number of input threads. Only the last thread finds the counter $= n$ and does not die— it continues as the sequel.

Instructions from the **`IFxx`** family are used at the top of a conditional expression: they test the integer source for the condition **`xx`** (such as **`EQ`** to 0, **`NE`** to 0, ...) and continue either on the left output thread (if true) or the right output thread (if false). `Merge` is used at the bottom of conditional expressions: a thread arrives at exactly one of its inputs, and continues. `Merge` may also be used at the top of loops to combine a back-edge with an initial input edge.

6.6 ALU and Local Memory Operations

A variety of two- and three-address instructions are available in the form shown in Figure 6. Opcodes include the usual integer, local-pointer and floating-point arithmetic. Local memory

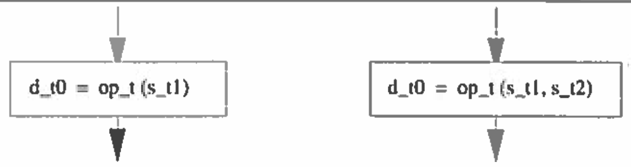


Figure 6: 2- and 3-address ALU operations in P-RISC graphs.

operations are shown in Figure 7. The `Load_local` instruction loads an lptr, given as a literal, into

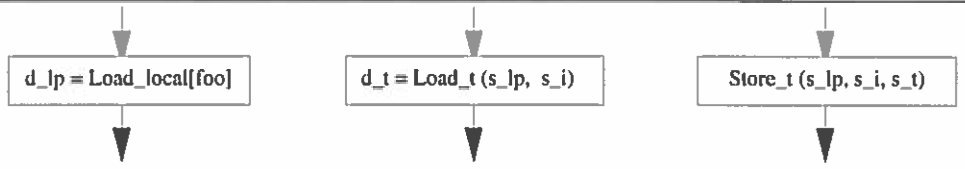


Figure 7: Local memory operations in P-RISC graphs.

its destination. The `Load` instruction takes an lptr and an integer offset and copies the contents of the indexed location to the destination. The `Store` instruction takes an lptr, an integer offset and a value and copies the value into the indexed location.

6.7 Global memory access

Global memory operations are shown in Figure 8. The `Load_global` instruction loads a global pointer, given as a literal, into its destination. The thread can subsequently load or store into the global destination using one of the remaining global operators, which are all so-called "split-phase" operations: the current thread initiates the global memory operation and continues, depicted by the dark output arrows. Later, the response from arrives and initiates a new thread, depicted by the light output arrows. The new threads begin with `Entry` instructions that receive the responses.

`Halloc` is really a macro but is sufficiently useful to be treated as an instruction. It takes two arguments, a size and a location hint, initiates a request to dynamically allocate a heap object of that size on that processor, and continues. Later, the heap allocator responds with a global pointer, which arrives at the `Entry` node and is loaded into its destination. `Halloc_empty` is similar except that it also ensures that all locations are reset to EMPTY. Clearly, it is unrealistic to expect that a single, fixed heap allocator will suit all applications, but it is useful to have a default one (much like `malloc` in C).

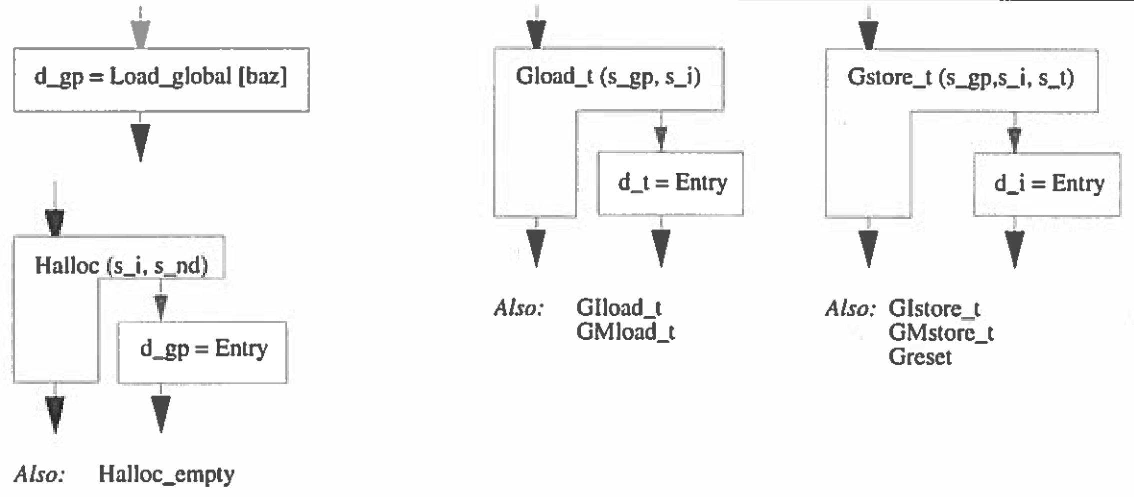


Figure 8: Global memory operations in P-RISC graphs.

`Gload` takes a gptr and an integer offset, sends a request to read the indexed global memory location, and continues. Later, the remote memory response arrives and initiates the thread at the `Entry` instruction, with the contents of the remote memory location loaded into the destination. This is a generalization of "pre-fetch" instructions found in some modern processors: in particular, there is no limit on the number of pending `Gloads`, and `Gloads` from arbitrarily distant, even unrelated, parts of the code can be interleaved.

`Gstore` takes a gptr, an integer offset, and a value; it sends a request to store the value into the indexed global location, and continues. Later, the remote memory responds with an integer status that initiates the thread at the `Entry` node, with the status value loaded into the destination. Status values are currently unspecified; the important thing is that the new thread is assured that the store has completed at global memory— this can be used to enforce various kinds of serialization. This is a generalization of "post-store" instructions found in some modern processors: work that does not depend on the serialization can be placed in the original thread while work that does depend on it can be placed in the new thread; there is no limit on the number of pending `Gstores`, and `Gstores` from arbitrarily distant, even unrelated parts of the code can be interleaved.

The `GIload/GIstore` and `GMload/GMstore` opcodes have the same interface as `Gload/Gstore`; the difference is in the operation performed at the remote memory location. They work with locations containing synchronized types. Recall that synchronized types also have a FULL/EMPTY state.

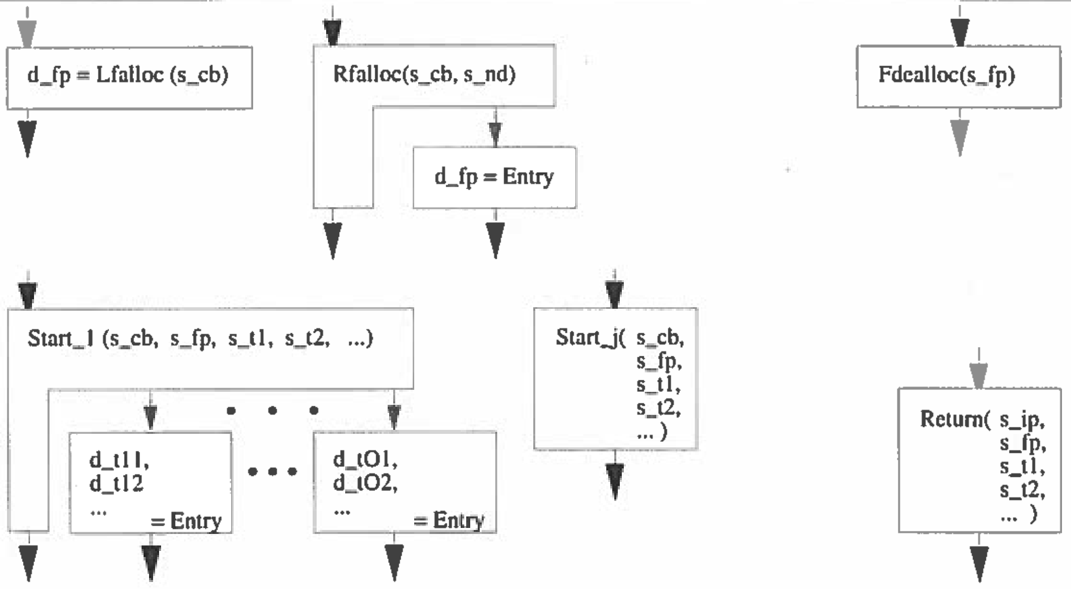
`GIload/GIstores` (Global, I-structure load/stores) are used for asynchronous, write-once situations. The location is initialized to EMPTY, a `GIstore` writes a value there and changes it to FULL, and `GIloads` wait until the location is FULL (*i.e.*, the `Entry` threads do not get initiated until the value is available). It is an error to `GIstore` to a FULL location. Thus, such locations can be used as write-once mailboxes. These semantics are called "I-structure" semantics, and correspond closely to the use of logic variables in Concurrent Logic Programming languages.

`GMload/GMstores` (Global, M-structure load/stores) are used for asynchronous multiple atomic reads and writes. The location is initialized to EMPTY, an `GMstore` is only allowed on an EMPTY location, at which point it writes the value there and changes it to FULL. An `GMload` waits until the location is FULL, reads it and atomically changes it to EMPTY. Thus, a location can be modified atomically by an `GMload`-*op*-`GMstore` sequence, no matter how long or complicated the *op*.

`Greset` can be used to reset a location to the EMPTY state. It is like `Gstore` except that it does not take a value argument.

6.8 Function calls

In general, a function can have I input threads and send results to O output threads in the caller. For most languages (*e.g.,* C, Fortran, Scheme, ...) we will usually have $I = O = 1$; however, for *non-strict* languages such as pH this is often not the case. Function linkage instructions are shown in Figure 9. The `Lfalloc` instruction takes a codeblock pointer, allocates a new frame for that



d_fp = Lfalloc (s_cb)
Rfalloc(s_cb, s_nd)
d_fp = Entry
Fdealloc(s_fp)
Start_1 (s_cb, s_fp, s_t1, s_t2, ...)
d_t11, d_t12 ... = Entry
d_tO1, d_tO2, ... = Entry
Start_j(s_cb, s_fp, s_t1, s_t2, ...)
Return(s_ip, s_fp, s_t1, s_t2, ...)

Figure 9: Function linkage operations in P-RISC graphs.

codeblock *locally, in the current stack* and returns the new frame pointer in the destination.

The `Rfalloc` instruction allocates a new stack (and a frame in that stack) which may be on another processor. It takes a codeblock pointer and an integer, initiates the request and continues. The integer is a processor number hint about where the new stack should be allocated. Later, the thread at the `Entry` instruction is initiated, depositing the new frame pointer (an gptr) its destination.

The `Start_1` instruction is the basic local function calling instruction. It initiates the first input thread of its codeblock argument with its `fp` argument, passing it some implicit arguments followed by the explicitly specified arguments `s_t1`, `s_t2`, ... The implicit arguments are the current frame pointer `FP` and the instruction pointers of the O `Entry` instructions that will receive the outputs of the function. The implicit arguments may be regarded as the O "return continuations" of this call. The corresponding thread in the callee will start with a matching instruction of the form

```
    d_fp   d_ip1  ...  d_ipO   d_t1  d_t2 ... = Entry
```

The `Start_1` instruction is an implicit fork, *i.e.,* the caller thread continues along the solid output edge after initiating the thread in the callee.

For codeblocks that have more than one input thread, the j'th additional thread ($j = 2, .., I$) is initiated by a `Start_j` instruction. The corresponding thread in the callee will start with a matching instruction of the form

```
d_t1 d_t2 ... = Entry
```

This instruction is also an implicit fork; execution of this thread continues after initiating the thread in the callee.

In the callee, each of the outputs is implemented by a `Return` instruction. It takes an instruction pointer (one of the "continuation" IPs sent over by the caller), a frame pointer (the "continuation" FP sent by the caller), and a list of results, and initiates the corresponding `Entry` node below the `Start_i` call. This instruction is also an implicit fork; execution of this thread continues after initiating the thread in the caller.

Finally, the `Fdealloc` instruction takes a frame pointer and deallocates the frame. This may occur either in the caller or in the callee, depending on compiler convention (our compiler does it in the callee).

6.9 An Example

In this section we look at various P-RISC Graph representations of a function `nleaves` that computes the number leaves of a binary tree:

```
nleaves  TNil           =  1
nleaves  (TCons x l r)  =  (nleaves l) + (nleaves r)
```

In order to start with a familiar case, Figure 10 shows a traditional sequential implementation (the recursive calls are done one after the other), in which the tree is allocated in local memory. We have coalesced linear, sequential strings of nodes (basic blocks) into larger boxes. At the top, the function receives its continuation frame pointer, continuation instruction pointer and a tree. It tests the tree for emptiness and branches accordingly. If false, it allocates a new (local) stack frame for itself (`fp1`), fetches the left subtree pointer (offset 1 from `t`), initiates the recursive call and halts. The recursive call returns, placing the leaf count of the left subtree into `nl`. It then fetches the right subtree (offset 2), initiates the second recursive call (reusing the frame `fp1`) and halts. This call returns with the right subtree leaf count in `nr`, which is added to `nl`. The frame `fp1` is then deallocated. Below the `Merge`, the function halts after initiating the continuation thread with the result.

We would like to outline how this can be implemented as efficienctly as, say, a traditional C implementation. First, we adopt a convention that `Entry` destinations are always allocated to a standard set of "argument-passing" registers, so that the `Entry` instruction itself becomes a no-op (the destinations are pre-loaded). Second, `Lfalloc` and `Lfdealloc` are simple increments and decrements of the current frame pointer. Third, the `Start_1-Halt` sequence can be implemented like a conventional call: load `FP` the current frame pointer, the address of the following `Entry` instruction and the argument `L` into the argument-passing registers, set FP to `fp1` and jump to `cb`. Similarly, the `Return-Halt` sequence just loads `n` into the first argument-passing register, sets FP to `cfp` and jumps to `cip`.

Figure 11 shows a parallel version of the function, with the tree allocated in global memory. At each node, we pursue the left subtree locally but initiate a parallel call to pursue the right subtree. On the false side of the `IFeq`, the thread initiates a global load for the right subtree, initiates a

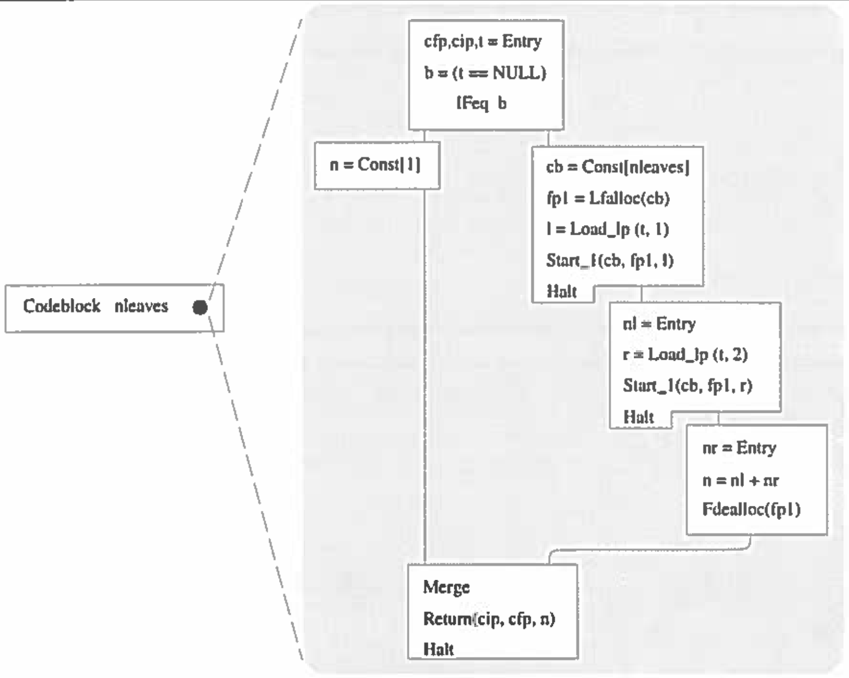


Figure 10: Sequential, local memory code for **nleaves**.

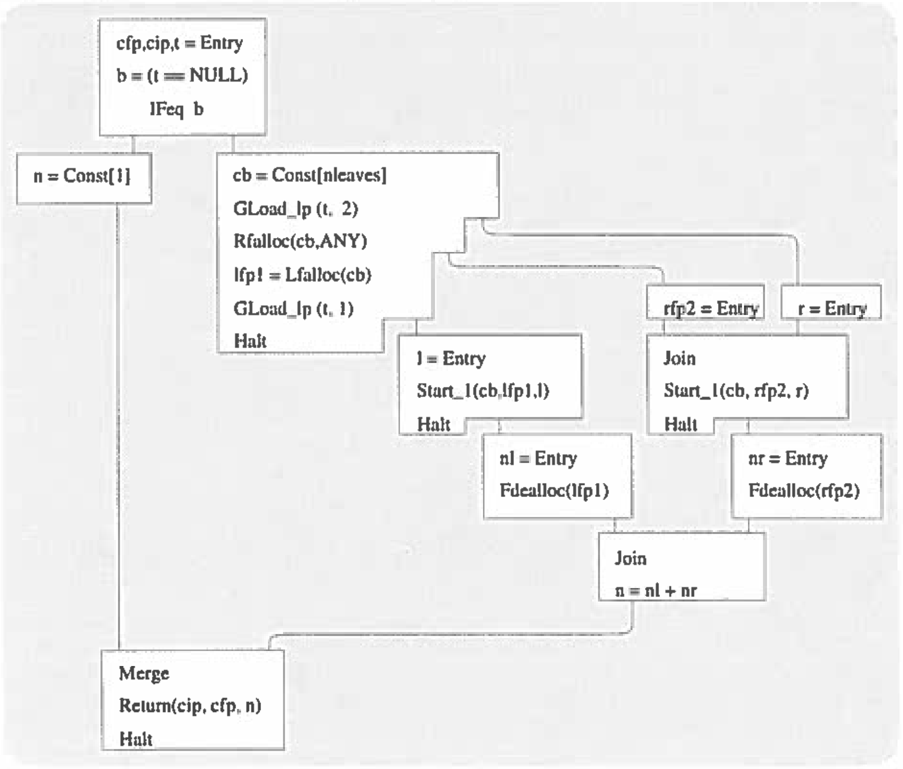


Figure 11: Parallel code for **nleaves**, tree in global memory.

new, remote stack allocation for traversing the right subtree, locally allocates a frame (`lfp1`) for traversing the left subtree, and initiates a global load for the left subtree. When the left subtree `l` arrives, we do a local recursive call to count its leaves. When that result `nl` arrives, we deallocate the local stack frame. When the remote frame `rfp2` and the right subtree `r` have both arrived, we initiate the remote recursive call to count its leaves. Whan that result `nr` arrives, we deallocate the remote stack. When both partial results `nl` and `nr` have arrived, we sum them and return the final result.

Finally, Figure 12 demonstrates non-strictness; it represents the function:

```
    f x = Cons x Nil;
```

The function has two inputs and two outputs. The first input (the "trigger") receives the two

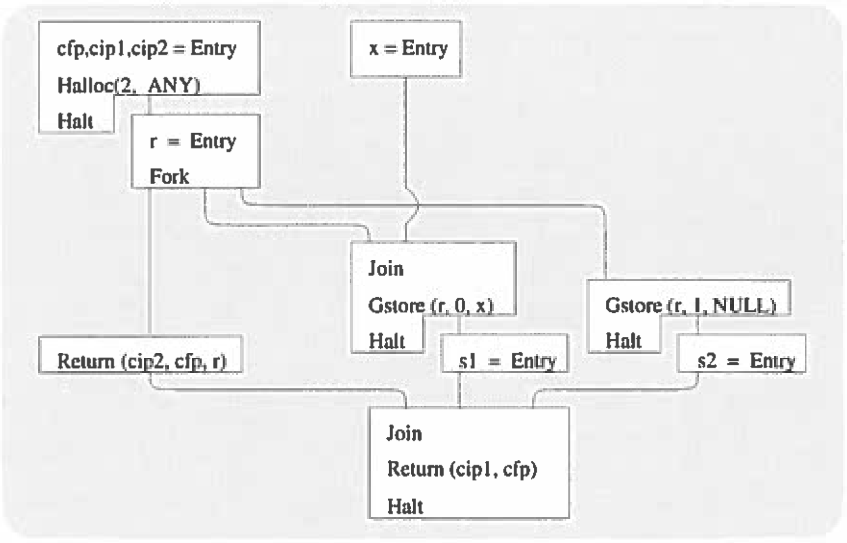


Figure 12: A non-strict function with 2 inputs and 2 outputs.

continuations— a frame pointer and two instruction pointers. It allocates a heap object of size 2, and returns the global pointer `r` to the thread at `cip2`. Meanwhile, we also initiate a global store of `NULL` into the second component of `r`. When `x` also arrives, it is stored into the first component of `r`. Finally, when both the stores have completed and `r` has been sent back, we return a "signal" to the other continuation (`cip1`).

Note that the caller does not have to wait until `x` is ready before calling this function— it can send the trigger earlier. In fact, it can even receive the result heap pointer `r` before `x` has been sent to the callee. In some applications, the `cip2` continuation may be able to begin some useful work already— it may not require the contents of the record immediately. The `cip1` continuation, on the other hand may be used to execute code that *does* need to read the contents of the record.

If we had used `Halloc_empty` and `GIstores` instead, then the caller can even initiate `GIloads` on the record as soon as the result pointer comes back. The `GIloads` simply block until the corresponding `GIstores` complete. Thus, threads can synchronize at the level of individual data elements.

7 P-RISC Assembler

P-RISC graphs are converted into P-RISC Assembler, which is a linear, textual representation. By and large, assembler instructions correspond exactly to their graph counterparts. However, certain things are now made explicit.

Labels: The pseudo-instruction `Label[L]` introduces a symbolic name `L` to which control can be transferred. Every `Entry` and `Merge`, and every true-destination of an `IFxx` must now be preceded by a label. `Merge` instructions can then be dropped, since they are no-ops.

Unconditional Jumps: The instruction `Jump[L]` transfers control to label `L`.

Conditional Jumps: The graph instruction `IFxx(s_i)` becomes `IFxx[L](s_i)` in P-RISC Assembler, where `L` is the label prefixed to the true-successor of the `IFxx` instruction.

Forks to labels: Given a `Fork` graph instruction with n outputs, we introduce labels for $n-1$ successors; convert the instruction into a series of $n-1$ `Fork[L]` instructions, each initiating one of those successors, and finally dropping through to the last successor.

Joins with explicit join counts: A graph instruction `Join` with n inputs becomes `Join[n](sd)`, where `sd` variable in the current frame; it is incremented by each incoming thread and compared to n. If `sd<n`, the incoming thread dies. The last thread, finding `sd=n`, continues. Note that a `Join` instruction does not have to be labelled— after replicating it on each incoming edge, the original `Join` can be converted into a `Merge` which, of course, must be labeled. For an n-way join, this prevents $n-1$ unnecessary transfers of control.

Explicit labels in split-phase operations: A graph instruction `Halloc(s_i,s_nd)` becomes `Halloc[L](s_i,s_nd)`, where `L` is the label of the `Entry` instruction that receives the new global pointer. Similaraly, other split-phase operations such as `Gload`, `Gstore`, `Rfalloc` and `Start_i` all get parameter lists containing the labels of the `Entry` instructions that await their results.

Codeblock declarations: A codeblock is declared using the form: `Codeblock glurph[n, L0,L1, ...]` where `n` is the size of the frame for the codeblock and `L0`, `L1` ... are the labels of the input `Entry` instructions.

8 Miscellaneous details

A detailed description of the principles of tranlation of pH source code to P-RISC graphs is given in [2] and [11], including dealing with non-strictness and higher-order functions. Those papers concentrate on retaining the full non-strictness of the source language. However, the examples in Section 6 should make it clear that the same source code can also be translated into sequential P-RISC graphs, or into strict, parallel P-RISC graphs. Our compiler supports all these options. We conjecture that strict, parallel P-RISC graphs (while retaining non-strict data structures) will offer the best compromise between parallelism and performance, for the forseeable future. Achieving efficiency while maintaining full non-strictness requires partitioning analysis (see, for example, [14]), which is a very difficult research problem, as hard as general strictness analysis.

A description of the mapping of P-RISC assembler to C is given in [9]. The translation is quite straightforward, and results in simple message-driven execution. Each node in the workstation farm endlessly executes incoming messages, each of which points at an `Entry` instruction. Execution of a thread beginning at an `Entry` continues until it hits a failing `Join` or a `Halt`. During a thread's execution, forked threads are pushed onto a scheduling stack, and split-phase instructions send out messages. When a thread ends, it pops the scheduling stack, executing any available previously-forked thread. When the scheduling stack is empty, *i.e.*, there are no more forked threads available, the processor goes to execute the next message, and so on. This model has also been implemented by Ellen Spertus on the MIT J-Machine [13], where she calls it the "flattened" implementation of TAM [3], as opposed to the "direct" implementation of TAM.[2]

9 Conclusion

We are nearing completion of a first implementation. Our focus, so far, has been on producing a complete implementation, which is a large undertaking, and so we have not yet spent much time on optimization. In particular, our current implementation does not yet do object-level cacheing. Our preliminary implementation runs on uniprocessors and Alpha workstation farms. We hope to release this system to the public in late Spring of 1994.

References

[1] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An Evaluation of Directory Schemes for Cache Coherence. In *Proc. 15th. Ann. Intl. Symp. on Computer Architecture, Hawaii*, May 1988.

[2] Arvind and R. S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Trans. Computers*, 39(3):300–318, March 1990.

[3] 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 *4th Intl. Conf. Architectural Support for Programming Languages and Operating Systems*, pages 164–175, April 1991.

[4] High Performance Fortran Forum. *High Performance Fortran: Language Specification, Version 1.0*, May 3 1993. Anonymous ftp: `titan.cs.rice.edu`.

[5] M. D. Hill, J. R. Larus, S. K. Reinhardt, and D. A. Wood. Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors. In *Proc. ASPLOS V, Boston, MA*, pages 262–273, October 1992.

[6] P. Hudak, S. Peyton Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. M. Guzman, K. Hammond, J. Hughes, T. Johnsson, R. Kieburtz, R. Nikhil, W. Partain, and J. Peterson. Report on the Programming Language Haskell, A Non-strict, Purely Functional Language, Version 1.2. *ACM SIGPLAN Notices*, 27(5), May 1992.

[7] T. Johnsson. Lambda Lifting: Transforming Programs to Recursive Equations. In *Springer-Verlag LNCS 201 (Proc. Functional Programming Languages and Computer Architecture, Nancy, France)*, September 1985.

[8] R. S. Nikhil. Id (Version 90.1) Reference Manual. Technical Report CSG Memo 284-2, MIT Lab for Computer Science, 545 Technology Square, Cambridge, MA 02139, USA, July 15 1991.

[9] R. S. Nikhil. A Multithreaded Implementation of Id using P-RISC Graphs. In *Proc. Sixth Ann. Wkshp. on Languages and Compilers for Parallel Computing, Portland, Oregon, Springer-Verlag LNCS 768*, August 12-14 1993.

[10] R. S. Nikhil. An Overview of the Parallel Language Id (a foundation for pH, a parallel dialect of Haskell). Technical report, Digital Equipment Corp., Cambridge Research Laboratory, September 23 1993.

[2]TAM, the Threaded Abstract Machine from Berkeley, is similar to P-RISC in many respects, but takes stronger positions on scheduling.

[11] R. S. Nikhil. The Parallel Programming Language Id and its Compilation for Parallel Machines. *Intl. J. of High Speed Computing*, 5(2):171–223, 1993. Presented at Workshop on Massive Parallelism, Amalfi, Italy, October 1989.

[12] S. L. Peyton Jones *et. al.* The glorious glasgow haskell compilation system, version 0.19, December 1993. Kit and documents by anonymous ftp from ftp.dcs.glasgow.ac.uk.

[13] E. Spertus, S. C. Goldstein, K. E. Schauser, T. von Eicken, D. E. Culler, and W. J. Dally. Evaluation of Mechanisms for Fine-Grained Parallel Programs in the J-Machine and the CM-5. In *Proc. 20th. Ann. Intl. Symp. on Computer Architecture, San Diego, CA*, May 17-19 1993.

[14] K. R. Traub, D. E. Culler, and K. E. Schauser. Global Analysis for Partitioning Non-Strict Programs into Sequential Threads. In *Proc. 1992 ACM Conf. on Lisp and Functional Programming, San Francisco, CA*, pages 324–334, June 22-24 1992.

Distributed Memory Implementation of KLIC

Kazuaki Rokusawa Akihiko Nakase Takashi Chikayama
rokusawa@icot.or.jp nakase@icot.or.jp chikayama@icot.or.jp
Institute for New Generation Computer Technology
1-4-28, Mita, Minato-ku, Tokyo 108, JAPAN

Abstract

This paper describes key issues of a distributed implementation of a concurrent logic programming language KL1 in the KLIC system. The implementation is realized without any changes in the core implementation of KLIC. External references are represented using *generic objects*. A new distributed unification scheme is investigated, which can cope with inter-processor reference loops. Incremental inter-processor garbage collection based on *weighted export counting* and quiescence detection (termination detection and global suspension detection) based on *weighted throw counting* are also presented.

1 Introduction

This paper describes key issues of a distributed implementation of KLIC [1], which is a portable implementation of a concurrent logic programming language KL1 [2]. They are external reference management, distributed unification and distributed goal management.

KLIC has a feature called *generic objects* which allows easy modification and extension of the system. Using generic objects, distributed implementation can be realized without changing the sequential core of KLIC. External references are represented as generic objects, and distributed unification operations are defined as methods of generic objects.

Creation of inter-processor reference loops cannot be avoided in distributed KLIC implementation, since exactly the same core is used which allows a binding of a variable to any data cell. To overcome this problem, a new unification scheme which can cope with reference loops is invented.

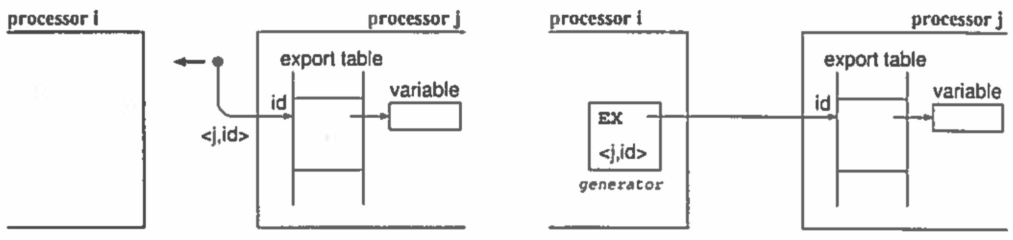
Termination detection and global suspension detection facilities are provided to support parallel programming and meta programming. Global suspension detection is newly introduced in KLIC implementation.

This paper is organized as follows. Section 2 describes external reference management and incremental inter-processor garbage collection. Distributed unification which can cope with inter-processor reference loops is presented in section 3. Section 4 presents a distributed goal management based on weighted throw counting.

2 External Reference Management

External Reference

When a message is sent to another processor and the message contains references to undefined variables or structures, the references across processors consequently appear; these



(1) Generating and Sending an exref ID. (2) Receiving an exref ID.

Figure 1: Generation of an External Reference

are the *external references*. The sender processor of the message *exports* the reference, and the receiver processor *imports* the reference.

An external reference could have been straightforwardly represented by the pair $< proc, addr >$, where *proc* is the processor number in which the referenced cell resides, and *addr* is the memory address of the cell. However, such an implementation causes a crucial problem; efficient local garbage collection (garbage collection within a processor) is impossible. If the locations of cells change as the result of local garbage collections, it must be announced to all processors that may have references to the cells.

In order to overcome this problem, each processor maintains an *export table* to register all locations of cells which are referenced from outside. The external reference is thus represented by the pair $< proc, ent >$, called the *exref ID*, where *ent* is the entry number of the export table. When the externally referenced cells are moved, the references from the export table entries are updated to reflect the moves, while the exref IDs are not affected.

External references are represented as generic objects within a processor. On receiving a message with an external reference, a generator object is generated, which is called *generator exref object* (figure 1). When a dereference request is made to the generator exref object, a read request message is sent to the referenced processor, and the object changes into a consumer object, which is called *consumer exref object* (see section 3.2).

Re-exportation

Since an exported cell is identified by its exref ID, distinct IDs are regarded as distinct cells even if they are identical. As an undefined variable or a structure may be exported to the same processor more than once, if the re-exported cell is given a new ID, redundant read/write request messages may be sent. To eliminate redundant inter-processor communication, an exported cell should not have more than one exref ID.

A hash table is attached to the export table to retrieve the same export table entry for the same exported cell. Also, there is a hashing mechanism for retrieving the same exref object from the imported exref ID. Thus, when a processor imports the same ID more than once, only one exref object is generated.

Inter-Processor Garbage Collection by WEC

Since export table entries cannot be freed by a local garbage collection, there must be some inter-processor garbage collection mechanism to free those entries that become garbage. To realize inter-processor garbage collection, the *weighted export counting* (WEC) scheme [3] is employed, which is based on the *weighted reference counting* (WRC) scheme [4, 5], a generalization of standard reference counting.

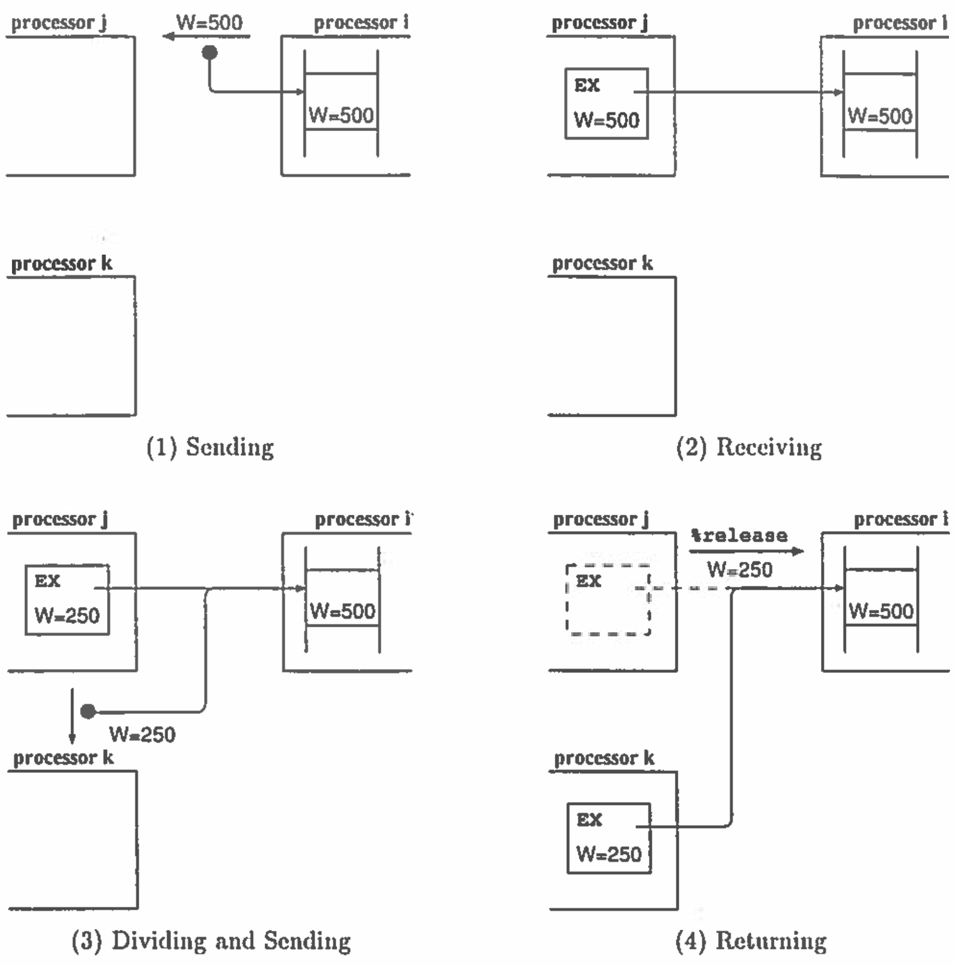


Figure 2: External Reference Management Under the WEC Scheme

The WEC scheme associates some weight (positive integer) to export table entries and external references (exref objects in processors and exref IDs in messages), so that the following invariant is true for every export table entry E :

$$weight\ of\ E = \sum_{x\ in\ references\ to\ E} weight\ of\ x$$

The above equality ensures that the weight of export table entry E reaches zero if and only if there is no external references to E.

When a new export table entry is allocated, the same weight is assigned to both the export table entry and the exref ID to be sent. When a processor receives an exref ID, it adds the weight assigned to the received ID to the weight of the exref object with the same ID. If there is no corresponding object, the processor generates a new object and sets its initial weight at the received weight.

When a processor sends an external reference, it assigns a weight to the exref ID and subtracts the same amount from the weight of the corresponding exref object. The new weight of the object and that assigned to the exref ID should both be positive, and the sum of the two weights should be equal to the original weight of the object.

When an exref object is released, its weight is returned to the corresponding export

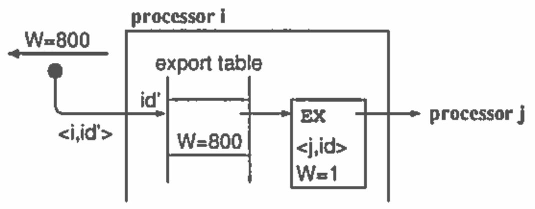
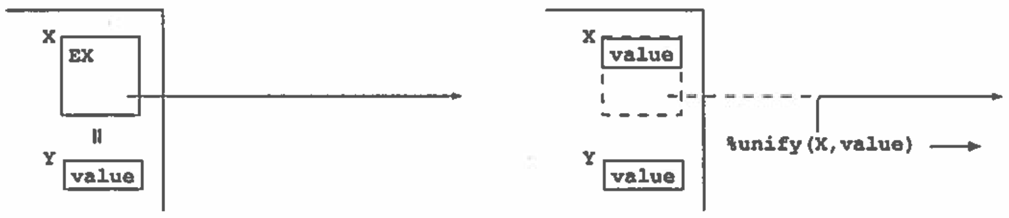


Figure 3: Indirect Exportation



(1) Before Unification (2) Overwriting and Message Sending

Figure 4: Unification between external reference and concrete value

table entry by a `%release` message[1]. On receiving a `%release`, the weight of the export table entry is decreased by the returned weight. If the weight of the export table entry reaches zero, the entry is freed. Figure 2 shows external reference management under the WEC scheme.

Indirect Exportation

When a weight of an exref object is one, the processor cannot send the external reference to another processor, because non-zero weight must be assigned to the exref ID and non-zero weight must remain also in the exref object after sending. In this situation, the processor performs *indirect exporting*; it registers the exref object itself and generates a new exref ID (figure 3).

3 Distributed Unification

This section describes unification involving external references and fetching a value from remote memory. Both are implemented based on message passing.

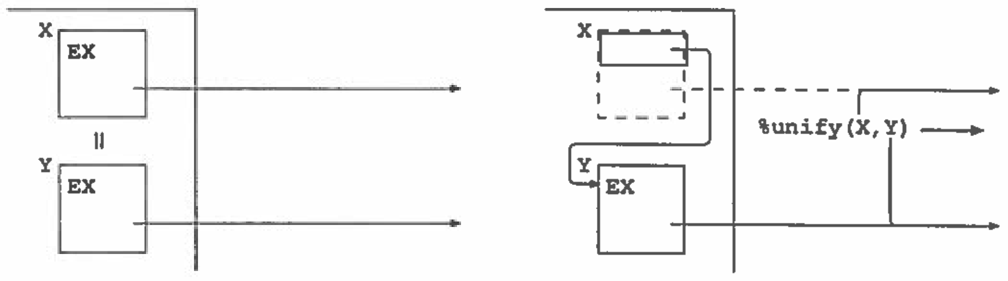
3.1 Unification Involving External References

Unification between an external reference X and a term Y is usually performed by sending the following message to the processor referenced by X.

```
%unify(X,Y)
```

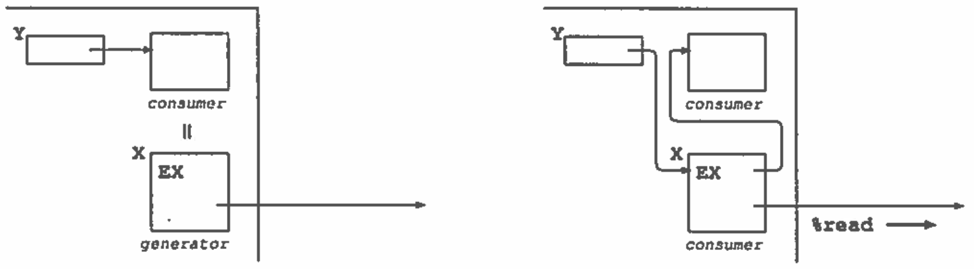
This is a request to unify the cell referenced by X with a term Y. In the sender processor of the message, X is overwritten by Y and the external reference is released; the weight attached to the reference is returned to the corresponding export table entry (see section 2).

[1]In this paper, message names are marked by prefixing it with a percent sign.



(1) Before Unification (2) Binding and Message Sending

Figure 5: When both are generator exref objects.



(1) Before Unification (2) Hooking and Message Sending

Figure 6: Unification between generator exref object and consumer object

The processor which receives the message performs the unification specified by the message. When the referenced cell is an external reference X', a `%unify(X',Y)` message is passed to the referenced processor. Details of unification operations are as follows.

Unification between external reference X and concrete value is done by sending `%unify(X,value)` message to the processor referenced by X. X is overwritten by the value (figure 4). The invoked *active unify* method of X performs these operations.

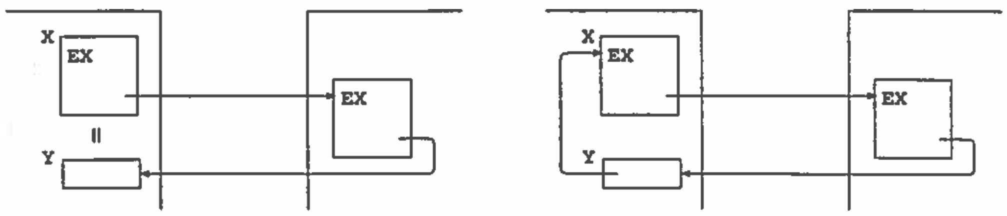
When both are generator exref objects (X and Y), `%unify` is sent to the processor referenced by one of the two generator objects (say X). X is overwritten by the reference to another object Y (figure 5). These operations are done by the invoked *active unify* method of X.

Unification between generator exref object X and consumer Y is performed as follows (figure 6).

1. The *generate* method of X is invoked and X changes into a consumer object.
2. The newly generated consumer object X is hooked to Y.

Unification between external reference X and variable Y is done by the core of KLIC, not by the method of X. The reference to X is simply bound to variable Y, which may create an inter-processor reference loop (figure 7)[2].

[2]In previous implementations on Multi-PSI and PIMs, many works has been done to avoid creation of reference loops, such as *safe/unsafe* attribute [3, 6, 7].



(1) Before Unification (2) Binding and Message Sending

Figure 7: Creation of Reference Loop

3.2 Fetching a Value From Remote Memory

When a goal reduction requires the value referenced by generator exref object X, the *generate* method of the object X is invoked. The method sends the the following read request message to the referenced processor to fetch a value.

```
%read(X,Ret)
```

`Ret` is a newly allocated external reference to X, which indicates the return address for response of the `%read` message. X changes into a consumer exref object and memorizes the return address. After sending a `%read` message, the suspended goal which is waiting for the value is hooked to consumer X (figure 8 (1) $\sim$ (2)).

On receiving a `%read` message, either of the followings is performed.

When the referenced cell is a variable or a consumer object, returning of value is suspended. A consumer object called a *reply object* which memorizes the return address is generated and hooked to the variable or the consumer object (figure 8 (3)).

When the referenced cell has a concrete value, an `%answer_value` message shown below is sent back to return the value (figure 8 (4)).

```
%answer_value(Ret,value)
```

When the referenced cell is a generator exref object X', a `%read(X',Ret)` message is forwarded to the processor referenced by X'.

On receiving an `%answer_value(Ret,value)`, if the cell referenced by `Ret` has a concrete value, a unification between the value of the referenced cell and the value carried by the message is performed. This occurs when unification has been made to the consumer exref object before arriving an `%answer_value`.

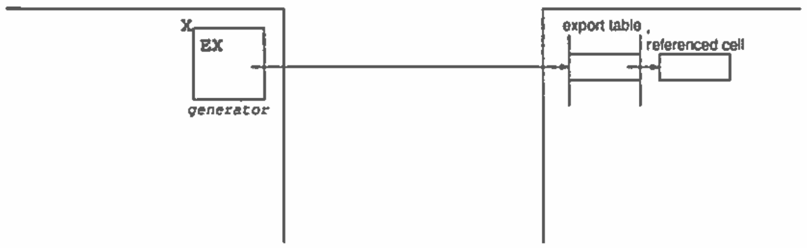
If the referenced cell is a consumer object, the consumer object is instantiated with the value carried and the following operations are performed to each of hooked goals and consumer objects (figure 8 (5) and figure 9).

Hooked goal is resumed.

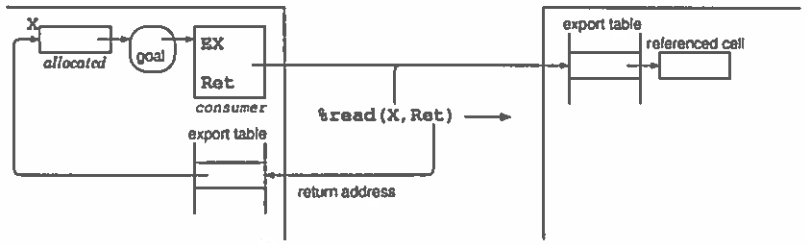
Consumer exref object with the **same return address** is released and a `%release` message is sent.

Consumer exref object with the **different return address** is released and a `%unify` message is sent. This is simular to the action when a consumer exref object and a concrete value are unified.

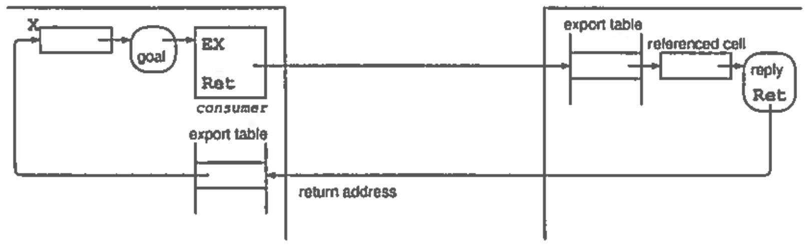
Reply consumer object is released and a `%answer_value` is sent.



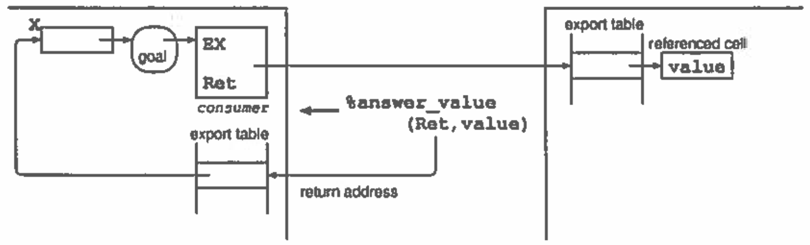
(1) Before Fetching



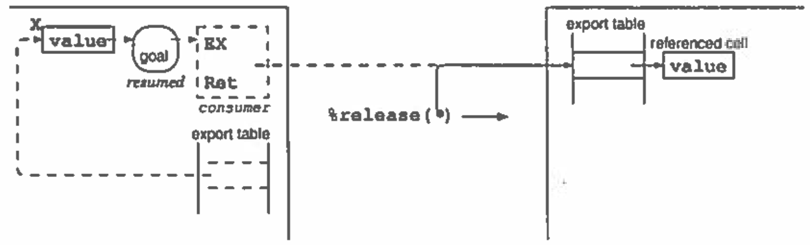
(2) Allocating a Return Address and Sending a %read message.



(3) Generating and Hooking a *reply object*.



(4) Sending Back an %answer_value message.



(5) Release of External Reference

Figure 8: Fetching a Value From Remote Memory

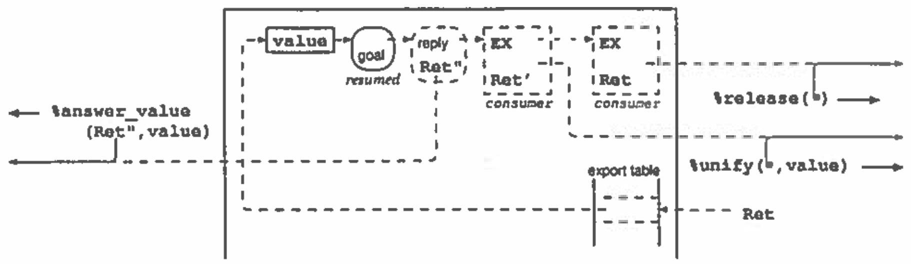
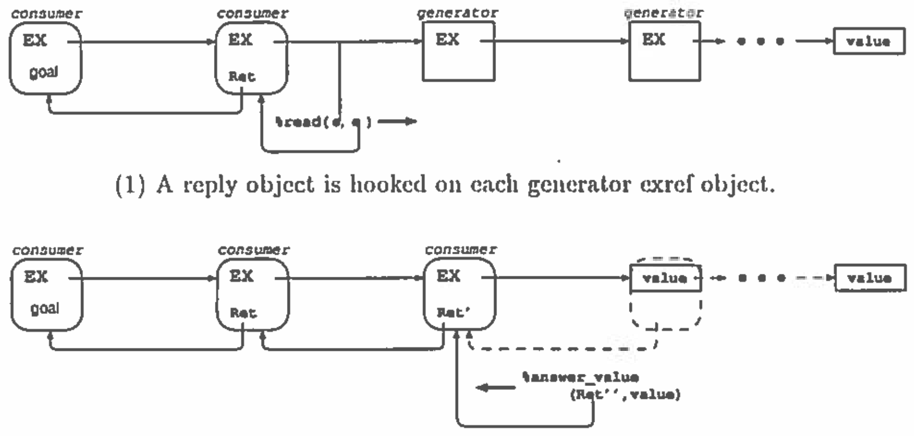


Figure 9: Operations when receiving %answer_value(Ret,value)



(1) A reply object is hooked on each generator exref object.

(2) An %answer_value message is returned along the same path.

Figure 10: *Hooking Reply Objects* Scheme

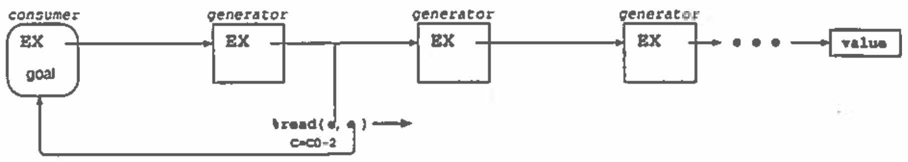
3.3 To Cope With Reference Loops

A *reference loop* is a closed chain of references. If there were a reference loop, the external reference on the loop would not have dereferenced results, and simply forwarding a %read message may not terminate.

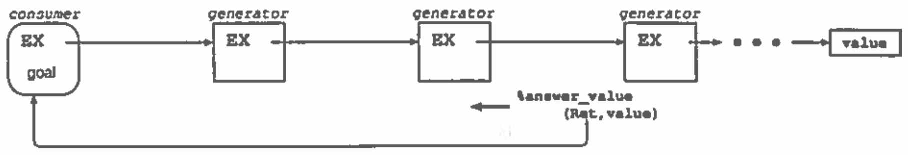
If a reply object is hooked to each generator exref object on forwarding, it is possible to check whether it goes into a reference loop or not. However, this *hooking reply objects* scheme has the following serious disadvantages (figure 10);

- It is necessary to hook useless reply objects even if a reference loop does not exist;
- An %answer_value message is returned along the same path as a %read message passed through.

To cope with reference loops, we have introduced a counter for a %read message which indicates the maximum number of forwarding. An original %read message is sent with the counter initialized. When a %read message arrives at a generator exref object, the value of the counter is checked. If the value is more than one, the %read is forwarded with the counter decremented. If the value is one, reply object is hooked and a %read message with the counter re-initialized and the return address reset to the external reference to the generator exref object.



(1) No reply objects are hooked.



(2) An %answer_value message is returned directly.

Figure 11: *Using a Counter* Scheme

Setting the initial value of the counter at the number of processors, it is expected that no useless reply objects are hooked and an %answer_value is always returned directly to the original consumer exref object (figure 11).

4 Distributed Goal Management

Global quiescence detection is a basic function in distributed processing. Unlike sequential computation, it is not trivial for distributed computation because of the difficulty in obtaining a consistent global state, particularly when there can be inter-processor messages in transit.

This section firstly describes a termination detection scheme called ***weighted throw counting*** (WTC), which is employed in previous KL1 implementations on Multi-PSI and PIMs [6, 7]. After that, global suspension detection scheme based on the WTC scheme is presented, which is newly introduced in a distributed KLIC implementation.

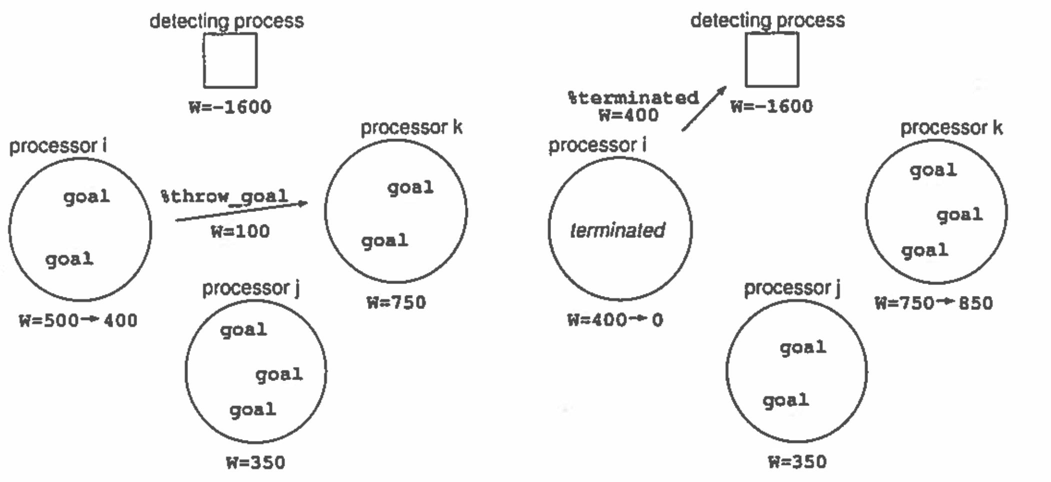
4.1 Computational Model

In a distributed environment, goals are distributed over processors. A processor may send a message to another processor, which carries a goal, a read request, a unify request, a dereferenced result, etc. As the sent message is delivered with arbitrary finite delay, messages may be in transit at a given time.

A processor can detect local termination; it is when all goals in it terminated and no suspended goals reside in it. When all processors locally terminate and there are no messages in transit, a distributed computation is considered globally terminated. To actually detect quiescence, a ***detecting process*** is placed on one of the processors.

4.2 Termination Detection

Termination detection is a difficult subject in distributed environments. Each processor can detect the local termination and can report it to the detecting process. However, even if a detecting process receives the report from all the processors, it is not sure that whole computation has terminated. There may be such messages in transit that carry goals, or bring dereferenced results and wake up suspended goals.



(1) Sending a message with weight (2) Receiving a message and returning a weight

Figure 12: The WTC scheme

To detect termination, the *weighted throw counting* (WTC) scheme [8] [3] is introduced. This scheme is an application of the *weighted reference counting* (WRC)[4, 5], which is a garbage collection scheme for parallel processing systems, and can efficiently detect termination without probing or acknowledgements[4].

In this scheme, the detecting process, each processor, and each message in transit have some *weight*. The weight of a message in transit and that of a processor are positive integers, while the weight of the detecting process is a negative integer. The WTC scheme maintains the invariant that:

> The sum of the weights is *zero*.

This ensures that the weight of the detecting process reaches zero if and only if all goals terminate, no suspended goals reside, and no messages are in transit.

When all goals in it terminate and no suspended goals reside, the processor locally terminates and sends a `%terminated` message to the detecting process to return its weight. On receiving `%terminated`, the detecting process adds the weight to its (negative) weight (figure 12). If the weight of the detecting process reaches zero, the global termination is detected.

4.3 Global Suspension Detection

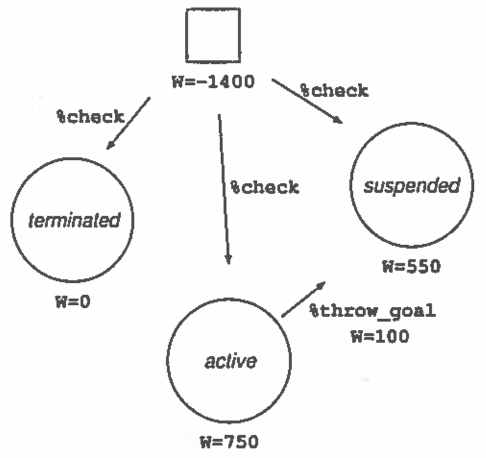
We define here the following stable state as *global suspension*.

> Although the whole computation has not completed yet, there exist neither active goals in processor nor messages in transit.

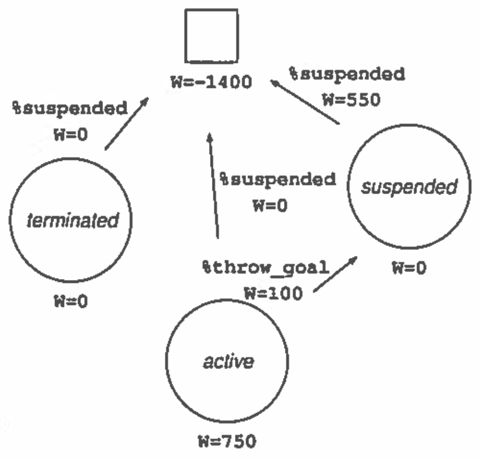
Global suspension detection is newly introduced in distributed KLIC implementation. In previous implementations on Multi-PSI and PIMs, perpetual suspension within a processor could be detected using MRB[11, 12] and local garbage collection[13]. However, it had no function for distributed environment. This section describes the basic idea of global suspension detection using the WTC scheme (figure 13).

[3]Essentially the same scheme named *Credit Distribution and Recovery* algorithm is presented in [9]. *Credit* in [9] corresponds to *weight* in the WTC scheme.

[4]Derivation of the WTC scheme from the WRC scheme is described in [10].



(1) Broadcasting a %check message



(2) Sending back a %suspended message

Figure 13: Global Suspension Detection

On start of detection, the detecting process allocates *ack counter*, and sets the initial values at the number of processors. The detecting process then broadcasts a %check message.

On receiving a %check, if all goals are suspended and no active goals reside, the processor sends back a %suspended message which carries all its weight. Otherwise, it sends bask a %suspended with no weight.

On receiving a %suspended, the detecting process adds the received weight to its (negative) weight, and decrements the *ack counter*.

Global Suspension Detection: When both values of its weight and the *ack counter* reach zero, the global suspension is detected; there are neither active goals in any processor nor messages in transit.

If the value of the *ack counter* only reaches zero, it is guaranteed that there are some active goal in some processor, or some message remains in transit.

5 Summary

Key issues of a distributed implementation of KLIC are described.

The implementation can be realized without slightly changing the sequential core of KLIC as the result of the followings.

- Generic objects are fully utilized; external references are represented as generic objects; operations of unification and dereference are defined as methods of generic objects.
- A new dereference scheme is invented which can cope with reference loops.

Global suspension detection scheme is also invented to support parallel programming and meta programming.

References

[1] Chikayama, T., Fujise, T., Sekita, D. : *A Portable and Efficient Implementation of KL1,* ICOT Technical Report, ICOT, 1994, to appear.

[2] Ueda, K. and Chikayama, T. : *Design of the Kernel Language for the Parallel Inference Machine,* The Computer Journal, Vol.33, No.6, pp.494–500, 1990.

[3] Ichiyoshi, N., Rokusawa, K., Nakajima, K. and Inamura, Y. : *A New External Reference Management and Distributed Unification for KL1,* New Generation Computing, Ohmsha Ltd., pp.159–177, 1990.

[4] Watson, P. and Watson, I. : *An efficient garbage collection scheme for parallel computer architectures,* Proc. PARLE : Parallel Architectures and Languages Europe, LNCS 259, Vol.II, pp.432–443, 1987.

[5] Bevan, D.I. : *Distributed garbage collection using reference counting,* Parallel Computing, Vol.9, No.2, pp.179–192, 1989.

[6] Nakajima, K., Inamura, Y., Ichiyoshi, N., Rokusawa, K. and Chikayama, T. : *Distributed Implementation of KL1 on the Multi-PSI/V2,* Proc. ICLP'89 : International Conference on Logic Programming, pp.436–451, 1989.

[7] Hirata, K., Yamamoto, R., Imai, A., Kawai, H., Hirano, K., Takagi, T., Taki, K., Nakase, A. and Rokusawa, K. : *Parallel and Distributed Implementation of Concurrent Logic Programming Language KL1,* Proc. FGCS'92 : International Conference on Fifth Generation Computer Systems, pp.436–459, 1992.

[8] Rokusawa, K., Ichiyoshi, N., Chikayama, T. and Nakashima, H. : *An Efficient Termination Detection and Abortion Algorithm for Distributed Processing Systems,* Proc. ICPP'88 : International Conference on Parallel Processing, Vol.I, pp.18-22, 1988.

[9] Mattern, F. : *Global quiescence detection based on credit distribution and recovery,* Inf. Proc. Lett., Vol.30, No.4, pp.195–200, 1989.

[10] Tel, G. and Mattern, F. : *The Derivation of Distributed Termination Detection Algorithms from Garbage Collection Schemes,* Proc. PARLE : Parallel Architectures and Languages Europe, LNCS 505, Vol.I, pp.137–149, 1991.

[11] Chikayama, T. and Kimura, Y. : *Multiple Reference Management in Flat GHC,* Proc. ICLP'87 : International Conference on Logic Programming, pp.276–293, 1987.

[12] Inamura, Y., Ichiyoshi, N., Rokusawa, K. and Nakajima, K. : *Optimization Techiniques Using the MRB and Their Evaluation on the Multi-PSI/V2,* Proc. NACLP'89 : North American Conference on Logic Programming, pp.907–921, 1989.

[13] Inamura, Y., Onishi, S. : *A Detection Algorithm of Perpetual Suspension in KL1,* Proc. ICLP'90 : International Conference on Logic Programming, pp.18–30, 1990.

Instant Replay Debugging of KLIC Programs

Kish Shen Steve Gregory

Department of Computer Science
University of Bristol
Bristol BS8 1TR, U.K.

Abstract

One problem with debugging (committed choice) concurrent logic programs is that their behaviour may be non-deterministic, so that successive executions of the same program may produce different results. We describe a scheme, based on the 'Instant Replay' scheme developed for more conventional parallel languages, that allows us to reproduce the execution behaviour of a concurrent logic program on subsequent executions, so that the execution may be examined for debugging purposes. Several properties of concurrent logic programming languages allow us to simplify our scheme greatly, and we believe it is feasible to use this scheme to debug large programs. We have demonstrated our scheme with KLIC, but it can be applied to other concurrent logic programming languages.

1 Introduction

One of the standard ways of debugging a sequential program is to re-execute the program when a bug occurs. This can be done because sequential execution is generally deterministic, so the bug would be reproducible. Moreover, generally, the behaviour is not affected by monitoring, so debugging tools can be used to examine the execution in detail to try to understand the bug.

This re-execution technique has formed the basis of most approaches to debugging sequential logic programs, with various debugging tools provided to ease the examination and controlling of the execution (*e.g.* [3]). Indeed, even algorithmic debugging, as conceived originally by Shapiro [8], and which is generally regarded as very different from traditional methods of debugging logic programs, incorporates the idea of repeatedly executing a program or parts of a program, and examining the output in a systematic way.

For a concurrent language executing in parallel, the re-execution approach to debugging has some serious drawbacks, because the execution of a concurrent program can be non-deterministic. That is, the actual execution, and its results, can vary between runs, because it depends on the relative timing of the various processes that are executing concurrently. Thus, if a bug occurs in one execution, a re-execution of the program may not in general be able to reproduce the bug. Moreover, as the exact behaviour is often very time-critical, tracing and monitoring the execution — operations that greatly perturb the timings — will make it even less likely that any non-deterministic bug will be reproducible.

Concurrent logic programming languages such as KL1 suffer from this problem, and this problem of non-deterministic bugs has generally not been taken into consideration in research work concerned with debugging (*e.g.* [9]). Tracing facilities of one form or another are usually provided with such systems (*e.g.* [2, 1, 10]), which in practice allow the programmer to remove most bugs by executing the program sequentially. However, bugs which only manifest themselves when executed in parallel, and which are not consistently reproducible, are much more difficult to eliminate. In this paper, we describe a method, based on the 'Instant Replay' method proposed by LeBlanc and Mellor-Crummey [6], which allows this difficulty to be overcome.

1.1 Instant Replay

Instant Replay was proposed as a way to overcome the problems of debugging concurrent parallel programs. It was developed within the framework of conventional concurrent languages, but it is in general applicable to most programming languages. The basic idea is to provide repeatable execution so that, when a program is being debugged, the same behaviour can be observed in as much detail and as often as desired. This is done by 'recording' the execution of a program, and 'replaying' the recorded execution should a bug occur. There is no risk of debugging tools perturbing the behaviour of the replay program, as its execution is controlled by the recording.

The important question is then what needs to be 'recorded'. At first sight, it might appear that a lot of information has to be recorded in order for the replay version to duplicate the behaviour of the recorded version: a concurrent program can be regarded as the execution of processes which may be created and destroyed dynamically, and which are able to communicate with each other. It may seem that both the evolution of the processes and the detailed behaviour of each process have to be recorded. This seems to imply a significant speed overhead, and the recorded execution will no longer closely resemble the original execution, so the possibility of not reproducing some of the bugs in the unrecorded version increases. Also, another possible disadvantage is that the recorded information generated can be huge, and storage of such information would be problematic.

Thus, it is essential for the recording process to be as efficient as possible, and this means recording the minimum of information. This is the objective of Instant Replay. LeBlanc and Mellor-Crummey made the observation that, in most concurrent programs, the execution of a process is actually deterministic, *if* its inputs are known. Inputs to a process come as communications from other processes, or are supplied initially. This means that for the whole execution, it is sufficient to know the order in which communications took place, *without* needing to know the contents of the actual messages that are transmitted: if the order is reproduced, the same communication will occur. This allowed the recording process to be much less expensive than in previous recording proposals, and so be usable in practice.

Instant Replay can be applied to a concurrent logic programming language such as KLIC to ease the problem of debugging non-deterministic bugs. Instant Replay will provide the ability to repeat equivalent executions of the program, and it can be integrated with existing debugging tools for debugging the program. One desirable aim is that the process should be as done as transparently as possible, so that the programmer does not have to explicitly specify how the program is to be recorded.

2 Instant Replay for KL1

2.1 Basic idea

A straight adaptation of the Instant Replay method to a concurrent logic programming language would mean recording the order in which all variable bindings are made, as this is the way one process communicates with another. However, concurrent logic programming languages have several properties that allow us to drastically simplify the recording process even further when compared to using Instant Replay on more conventional languages:

- Single assignment property: variables in KLIC can only be assigned to once, and variables are the only means of communicating between different processes (goals) in KLIC.
- A variable cannot be assigned conflicting bindings without the whole program failing.
- 'Don't care' non-determinism [5]: only one clause of a predicate is allow to proceed beyond the guard when it is called.

These properties together mean that the only source of non-deterministic behaviour in a KLIC (and other concurrent logic programming languages in general) program which has user-visible effects is the actual clause to which each individual goal commits during execution. The other source of non-deterministic behaviour, the order in which goals within a clause is executed, does not (in general) have any user-visible effect on the computation, if the same clauses are being committed to. If the program succeeds, the same variable bindings must be made. Deadlock is independent of the order of execution of the goals, as is failure, except that the latter may be detected earlier or later. Behaviour is also not affected in cases where the program loops infinitely. Thus, it is sufficient to record which clauses are being committed to and, if these commitments are repeated during replay, the replay program will have equivalent behaviour to the program in which the recording was done, independent of the order of execution of individual goals. Successes (with the same answers), loopings, failures or deadlocks in the recorded program will be faithfully repeated during replay. Most importantly, unlike the original or recorded executions, where the behaviour is time-dependent, the replay execution has no such dependence, and can thus be monitored in as much detail, and repeated as often, as desired.

2.2 Our implementation: basic scheme

This recording can be done at a low level, but it could also be done at the source level, which requires no changes to the implementation, as only a source-to-source transformation is needed. Our expectation for the scheme was that it would be sufficiently efficient, even if done at the source level, so that the overheads introduced will not significantly perturb the behaviour of the system.

Basically, the original program is transformed into two versions: one for record, and the other for replay. An extra argument is added to the head and every (non-built-in) body goal of the program. This argument is a data structure that represents the commitments made by the goals. This structure is built by the record execution; during replay, the structure from a previous record execution is used to control the execution so that the same commitments are made. As discussed, this is sufficient to ensure that the replay execution is equivalent to the recorded execution. The transformed programs (both record and replay) are referred to as being *monitored.*

As the transformation is done at the source level, only user defined predicates can be modified and monitored in this way. Fortunately, most built-ins are deterministic: that is, given the same input, they will have the same behaviour whenever they are executed, so there is no need to monitor them.

The transformation of the source program is best illustrated by an example. Consider one clause from a KLIC program that finds the derivative of an expression:

```
d(U+V,X,D) :- D=DU+DV, d(U,X,DU), d(V,X,DV).
```

This clause is transformed into the following for recording:

```
d(U+V,X,D,_A) :- _A = c1(_B,_C), D=DU+DV, d(U,X,DU,_B), d(V,X,DV,_C).
```

Extra arguments are added to the head of the clause, along with each non-built-in body goal. An extra body goal, `_A = c1(_B,_C)`, is added to collect the structure built by calling the body goals, and pass it via the head argument (`_A`) to whatever clause called this predicate. The name of the structure being built (`c1` in this case) is different from the other clauses in this predicate (`d/3`) in order to identify the clause that is being committed to. An extra goal has to be added in KLIC because all arguments in the head of a clause are input arguments. Thus, the structure that is built is a tree, with each goal adding a node to the tree. Each node identifies which clause is being committed to. We refer to this data structure as the *trace tree.*

The trace tree generated by the record program is used by the replay program to control its execution. The replay transformation for the example clause is:

```
d(U+V,X,D,c1(_B,_C)) :- D=DU+DV, d(U,X,DU,_B), d(V,X,DV,_C).
```

Each node of the trace tree forces the system to commit to the same clause as the recorded program did because the extra argument (`c1(_B,_C)`) will only match this one clause. In this way, the replay version of the program will behave in an equivalent manner to the recorded version.

Figure 1 shows a more complete example of the record and replay transformation: the whole derivative program, along with the record and replay versions of the same program, is shown.

```
% original

test :- d(Expr=(x+1)*(x^2+2)*(x^3+3),x,D).

d(U+V,X,D)  :- D=DU+DV,  d(U,X,DU), d(V,X,DV).
d(U-V,X,D)  :- D=DU-DV,  d(U,X,DU), d(V,X,DV).
d(U*V,X,D)  :- D=DU*V+U*DV,  d(U,X,DU), d(V,X,DV).
d(U/V,X,D)  :- D=(DU*V-U*DV)/V^2,  d(U,X,DU), d(V,X,DV).
d(U^N,X,D)  :- integer(N) | N1 := N-1, D=DU*N*U^N1, d(U,X,DU).
d(exp(U),X,D)  :- D=exp(U)*DU,  d(U,X,DU).
d(log(U),X,D)  :- D=DU/U,  d(U,X,DU).
d(X,X,D)  :- D=1.
otherwise.
d(C,_X,D)  :- D=0.

%record
test :- test(_A), builtin:print(_A).

test(_A)  :- _A = c1(_B), d(Expr=(x+1)*(x^2+2)*(x^3+3),x,D, _B).

d(U+V,X,D,_A)  :- _A = c1(_B,_C), D=DU+DV,  d(U,X,DU,_B), d(V,X,DV,_C).
d(U-V,X,D,_A)  :- _A = c2(_B,_C), D=DU-DV,  d(U,X,DU,_B), d(V,X,DV,_C).
d(U*V,X,D,_A)  :- _A = c3(_B,_C), D=DU*V+U*DV,  d(U,X,DU,_B), d(V,X,DV,_C).
d(U/V,X,D,_A)  :- _A = c4(_B,_C), D=(DU*V-U*DV)/V^2,  d(U,X,DU,_B), d(V,X,DV,_C).
d(U^N,X,D,_A)  :- integer(N)  | _A = c5(_B) N1 := N-1, D=DU*N*U^N1, d(U,X,DU,_B).
d(exp(U),X,D,_A)  :- _A = c6(_B), D=exp(U)*DU,  d(U,X,DU,_B).
d(log(U),X,D,_A)  :- _A = c7(_B), D=DU/U,  d(U,X,DU,_B).
d(X,X,D,_A)  :- _A = c8, D=1.
otherwise.
d(C,_X,D,_A)  :- _A = c9, D=0.

%replay
test :- builtin:read(_A), test(_A).

test(c1(_B))  :- d(Expr=(x+1)*(x^2+2)*(x^3+3),x,D, _B).

d(U+V,X,D,c1(_B,_C))  :- D=DU+DV,  d(U,X,DU,_B), d(V,X,DV,_C).
d(U-V,X,D,c2(_B,_C))  :- D=DU-DV,  d(U,X,DU,_B), d(V,X,DV,_C).
d(U*V,X,D,c3(_B,_C))  :- D=DU*V+U*DV,  d(U,X,DU,_B), d(V,X,DV,_C).
d(U/V,X,D,c4(_B,_C))  :- D=(DU*V-U*DV)/V^2,  d(U,X,DU,_B), d(V,X,DV,_C).
d(U^N,X,D,c5(_B))  :- integer(N)  | N1 := N-1, D=DU*N*U^N1, d(U,X,DU,_B).
d(exp(U),X,D,c6(_B))  :- D=exp(U)*DU,  d(U,X,DU,_B).
d(log(U),X,D,c7(_B))  :-  D=DU/U,  d(U,X,DU,_B).
d(X,X,D,c8)  :- D=1.
otherwise.
d(C,_X,D,c9)  :- D=0.
```

Figure 1: Example record and replay versions of a KLIC program

2.3 Improving the basic scheme

2.3.1 Detecting determinism

In the basic scheme outlined, every call to a goal that is not a built-in is recorded. This can generate a very large tree for a large program. Although the overhead introduced by the extra argument is small, the tree being built during the record phase has to be kept in memory, and if it becomes sufficiently large, its very size will induce extra speed overheads because of extra swapping/paging and garbage collection, which might significantly affect the behaviour of the recorded program, so that it no longer closely resembles the original program. In the worst case, the record program may not be executable at all, whereas the original program can be executed. It is thus desirable to reduce the size of the tree as much as possible, in order to minimize its impact on memory management.

The size of the tree can be reduced by not monitoring the execution of every goal. Intuitively, if the candidate clause for a particular goal can be determined uniquely by the head matching and guard execution, then there is no need to record this commitment, as it is deterministic. We call this form of determinism *locally deterministic*: a goal is locally deterministic if at most one of the clauses of the predicate representing the goal is applicable whenever the goal is called. That is, all but one (or none) of the clauses must fail in their head and guard unifications.

Unfortunately, a locally deterministic goal may still need to be recorded, because it may call other goals which need to be recorded. In addition, if one clause of a predicate needs to be recorded, then all the clause heads need to be transformed, even for clauses whose body goals do not need to be recorded.

We refer to a goal as being *totally deterministic* if it does not have to be recorded. A goal is totally deterministic iff it is locally deterministic *and* does not make any calls, directly or indirectly through its subgoals, to any goal that is not locally deterministic. Note that this applies to all clauses of a goal, and all clauses of any subgoals that are called.

Only goals which are not totally deterministic have to be converted for record and replay. To determine which goals in a program are totally deterministic, we perform a simple abstract interpretation of the program. We define the following abstract domain, arranged as a complete lattice as shown in Figure 2:

$$D_\alpha = \{\bot, [local\ det], [total\ det], \top\}$$

The concretization function is:
$\gamma : D_\alpha \longrightarrow G, G =$ set of goals appearing in clauses of the program

$$\begin{aligned}
\gamma(\bot) &= \emptyset \\
\gamma([local\ det]) &= \{locally\ deterministic\ goals\} \\
\gamma([total\ det]) &= \{totally\ deterministic\ goals\} \\
\gamma(\top) &= G
\end{aligned}$$

The abstraction function α can be easily derived from the concretization function. Note that $\top$ is the most general abstraction, as any goal can be abstracted to $\top$.

The abstract interpretation is done as a two stage bottom-up analysis: first we assume that all goals are locally deterministic, and then determine which are not. Goals which are not locally deterministic are moved to $\top$. Next, the goals which are locally deterministic are assumed to be totally deterministic, and we then promote the status of goal in $\top$, so that any goals which were assumed to be totally deterministic, but call goals in $\top$, are also moved to $\top$. Fixpoint is reached when all goals that directly or indirectly call any goal that is not totally deterministic are all classified as $\top$.

This information is then used by the converter, so that any goals that are totally deterministic are not converted. Consider the derivative program of Figure 1. As both the `d/3` and `test/0` goals are locally deterministic, it

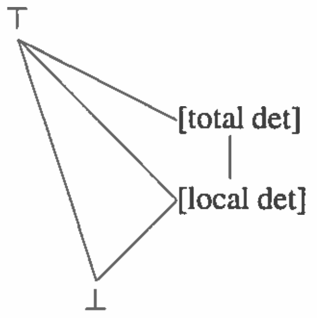


Figure 2: Determinism domain

means that the whole program does not need to be converted at all.

The aim of our abstract interpreter is to classify as many goals as possible as totally deterministic. However, in our first implementation, we do not treat each call to a predicate individually. Instead, we perform our analysis at the predicate level, so that all calls to a particular predicate are treated in the same way. This may lose some precision, because one goal for the predicate may be totally deterministic, but another goal for it may not, and the worst-case assumption of the predicate being not totally deterministic has to be made. A good example of this is the derivative program: `d/3` is only locally deterministic if the second argument is atomic. If `d/3` is called somewhere else with the second argument not atomic (which fortunately it is not), then the worst-case assumption of `d/3` being not locally deterministic must be made for all calls of `d/3`. The advantage with our implementation is that it greatly simplifies the analysis, and it is sufficiently accurate to be very useful. Most predicates tend to be used in a very similar fashion in KLIC, so that, if one goal for it is deterministic, others are usually deterministic also, so the gain for being more precise tends to be small.

The quality of the analysis depends crucially on how well the system is able to uncover predicates which are locally deterministic. Currently, we use a very simple scheme where the clause heads of a predicate are checked to see if any pair of them are mutually unifiable. If they are, then the clause heads are not mutually exclusive, and the predicate is assumed to be ⊤. This is a safe operation in KLIC, because no output bindings are allowed in the head, so that unlike Prolog, no mode analysis of the arguments of a goal has to be performed. We plan to increase the sophistication of our analyser by examining the guards of clauses whose heads can be unified. The determinism analyser of Andorra-I [7] should be useful for this purpose.

An example of the benefit of performing the determinism analysis is shown in Figure 3. This is a fragment of a program for solving the N-Queens puzzle. If the analysis is able to determine that both `gen/2` and `count/2` are totally deterministic, then these two predicates do not have to be converted, and the actual conversion that is done is shown in the figure.

In addition to the analyser uncovering goals which are locally deterministic, we also allow the programmer to specify that any predicate is locally deterministic. This is useful for several reasons — for example, it allows the user to help the analyser in cases where it is difficult to uncover the determinism automatically. An example of this is the derivative program of Figure 1: as already mentioned, `d/3` is locally deterministic only because the second argument is bound to an atom. Our current analyser is not sophisticated enough to uncover the determinism in this case, and no matter how sophisticated the analyser, there may always be cases where it cannot uncover the determinism, so it is thus useful for the user to be able to provide guidance to the analyser.

User annotated locally deterministic predicates also have another use: to selectively turn off the monitoring of parts of a program in a controlled way. In this case, predicates which are actually *not* locally deterministic are annotated as locally deterministic, and, as long as the predicate does not call any other goals which are not locally deterministic, the predicate need not be monitored. This is a much more controlled and safer way of removing a predicate from monitoring than allowing the user to specify that a predicate is totally deterministic, because of

```
go(N,M) :-
     gen(N,L),
     queen(L,[],[],X,[]),
     count(X, M).

%record
go(N,M,_A) :-
   _A = c1(_B),
     gen(N,L),
     queen(L,[],[],X,[], _B),
     count(X, M).

%replay
go(N,M,c1(_B)) :-
     gen(N,L),
     queen(L,[],[],X,[], _B),
     count(X, M).
```

Figure 3: Example conversions with determinism analysis

```
go(N,M,_B) :-
     gen(N,L),
     queen(L,[],[],X,[], _B),
     count(X, M).
```

Figure 4: Example conversions with removal of redundant nodes

the possibility that the goal may call other goals that are not locally deterministic.

The user can use this method to turn off the monitoring of regions of the program that are not considered to be problematic, thus further reducing the size of the trace tree. This is particularly useful for goals which actually return a deterministic answer, although some subgoals called by the goal are actually non-deterministic. One example of this is the summing of the numbers in the list produced by the non-deterministic merging of two lists of numbers. The merging of the list is non-deterministic, but not the total of the summation. In such a case, if the user knows there are no bugs inside the merge and sum routines, recording can be turned off by specifying that merge is locally deterministic.

2.3.2 Removing redundant nodes in trace tree

After removing the monitoring of totally deterministic predicates, the trace tree can still be reduced further in size, because it contains some redundant nodes. Normally, if a clause is a member of a predicate which is locally deterministic, but contains body goals which are not totally deterministic, then the clause must be recorded. However, if only one of the body goals is not totally deterministic, there is no reason to monitor the commitment itself, although the non-totally deterministic body goal still has to be monitored. No node of the tree needs to be created for the clause, instead, the extra argument is just passed to the non-totally deterministic body goal.

This improvement can be illustrated using the N-Queens fragment in Figure 3. It can be seen that `go/2` is locally deterministic, as it has only one clause. It also contains only one goal, `queen/5`, that has to be monitored. Thus, the local node can be removed, resulting in identical converted code for the record and replay versions, as shown in Figure 4.

This reduction in the trace tree size can be applied whether or not determinism analysis is done. However, it is particularly useful when used in conjunction with the determinism analysis, because the determinism analysis is able to remove many goals from being monitored, so it is more likely that there is only one goal in the body that is monitored. In fact, many KLIC programs are largely deterministic, with only a few local areas of non-determinism. If these non-deterministic goals are called after a long chain of otherwise deterministic goals, this optimization would remove many intervening redundant nodes.

2.4 Expected method of use

With the availability of Instant Replay for the KLIC system we expect that, during the initial stages of development of a program, the record mode is enabled all the time. Then, as the programmer becomes confident that a particular section of the code is bug free, recording is turned off, until the whole program is no longer recorded in the 'production' version. If the cost of recording is low, we expect to be able to catch most time-dependent (and usually unrepeatable) bugs during the development. In addition, when the determinism analysis is performed, the user can examine which goals the program thinks are not locally deterministic, and annotate any such goals which are actually locally deterministic. Examining the result of the determinism analysis has another advantage: it allows the user to eliminate unintentional non-determinism.

Furthermore, in conjunction with other tools, we expect that 'performance debugging' can be performed to tune the performance of a program: the execution of a program is recorded, and then replayed and observed to see how improvements can be made. This will be very useful for improving the performance of production versions of programs.

3 Other issues

3.1 Non-deterministic built-ins

One assumption of our approach is that built-in predicates are deterministic. This is in fact generally true of KLIC, with the exception of some of the I/O primitives. I/O should generally not present a problem, because the user should be careful of how they are used, given that there is freedom in their execution order. However, it may still be desirable to ensure that the same behaviour is obtained in the replay phase, as otherwise their non-deterministic nature may cause the replay program to fail. In addition, general methods for dealing with non-deterministic built-ins will be useful, as they may be provided in new versions of KLIC (and also other concurrent logic programming languages).

Built-ins can appear in both the guard and the body of a clause. Those appearing in a guard present no problem: provided they generate no variable bindings, their only purpose is to select a candidate clause. In the replay version of the program, the choice of clause is fixed by the trace tree, so the guard goals can be completely removed from the program.

Many non-deterministic built-ins (in the body) give different results according to the binding state of their arguments, that is, whether variables in their arguments are bound. Examples of such built-ins include `print` and `var` (though the latter is not currently available in KLIC). To preserve the behaviour of these built-ins on replay it is necessary, and sufficient, to somehow record the binding state at the time the goal was executed. One way to do this might be to use the `copy_term` of Prolog, which is not currently available in KLIC. This would be used to copy the goal at record time *before* it is executed. (The copy of the goal is executed, not the original, which may become further instantiated.) The copied goal is then stored as a node in the trace tree, where it can be executed at replay time. An alternative method is to replace such goals by calls to suitably defined user predicates which have different clauses for different cases of variable binding. For example, `print(X)` could be replaced by `nd_print(X)`, which is defined in Figure 5 (this needs to be extended to handle structured terms). The behaviour of the substituted goal will be reproduced by the usual method of recording the clause that it commits to.

```
    nd_print(X) :- atomic(X) | print(X).
    alternatively.
    nd_print(X) :- print('_U').
```

Figure 5: `nd_print`

Other non-deterministic built-ins, such as the `readint` primitive, can be converted as shown in Figure 6. In this case, the record version adds a node of the trace tree that is actually the integer being read in, instead of commitment information. The read is not needed in the replay version, as the original value read in is supplied by the trace tree.

```
% Original
foo(X) :- builtin:readint(X).

%record
foo(X, _A) :- readint(X, _A).

readint(X, _A) :- _A = X, builtin:readint(X).

%replay
foo(X, _A) :- _A = X.
```

Figure 6: Dealing with input built-ins

These methods are quite general, and can be applied to other built-ins which might return non-deterministic results, as long as sufficient meta-programming support predicates are available. It cannot deal with all cases, in particular, if a built-in affects the global state is some way (*e.g.* some form of global variables, or assert/retract), then this method would not work. However, such built-ins are probably not very suitable for a distributed concurrent language, so are unlikely to appear in a language such as KLIC.

3.2 Partial transformations and low-level support

Another problem with the method is that, with determinism analysis, only some of the predicates of the program are transformed. There is a possibility that the name of a transformed predicate will clash with an existing untransformed predicate. The transformation program can be made to automatically deal with this by renaming one of these predicates if it arises.

Currently, we convert programs quite independently of the KLIC system, and we perform the monitoring at a high-level. Although, as we shall show shortly, the monitoring does not in practice introduce a significant overhead, it will nevertheless be more efficient if the monitoring is more integrated with the KLIC system, and is done at a lower level, without doing a source-to-source transformation. Additional advantages for a lower level monitoring is that the monitoring will be transparent to the user, as the user need not see a converted program. In addition, a lower level monitoring will also be able to cope with failures, deadlocks and infinite loops better than our current system. Although all the information needed to reproduce the failures, deadlocks and infinite loopings are available in the trace tree in our approach, we currently have no easy way to access this information. A lower level approach will be able to access this information directly when the system detects a failure or deadlock, or when the user interrupts the execution because infinite looping is suspected.

4 Initial results

As a parallel version of KLIC is not yet available, our initial evaluation of the system consists mainly of examining the sizes of the trace trees that are generated, and how their generation affects execution in the record phase. We also tested our system by simulating a parallel environment by imposing different priorities on the goals in the record and replay versions. Normally, the replay version of the program would behave differently because of the different priorities but, because of the trace tree, it should reproduce the behaviour of the record version, irrespective of any changed priorities.

As we already noted, for a given program and query, the smaller the trace tree, the less the record version of the program is perturbed, and the closer it reflects the behaviour of the original program. We measure the size of the trace tree by simply counting the number of characters in its printed representation. We feel that this is a reasonably accurate and simple method of measuring relative sizes of terms: different clauses of a predicate are given an unique id by numbering in a left-to-right textual order. Therefore, unless there are 10 or more clauses to a particular predicate, this method of measuring relative sizes should be quite accurate, and even with cases of predicates with many clauses, the error would still be relatively small.

Most of the example programs provided with the KLIC system are deterministic. That is, all their predicates are totally deterministic. With the help of user annotation, we are able to avoid converting any goals and, effectively, no trace tree is generated at all.[1] However, with the unoptimized basic scheme where all non-built-in goals are converted, the size of the tree generated can be relatively substantial. For example, for the $5\times4\times3$ puzzle program [11] (a deterministic program), the size of the trace tree is just over a million characters, compared to a size of 0 if determinism analysis is done. The execution time of the record version is 3.5 ± 0.4 seconds, compared to 2.4 ± 0.6 seconds for the original unconverted program (both figures are averages and standard deviations for 3 timings). Thus, even if all goals are monitored, the execution time is not dominated by the extra overheads, although they can be significant. We expect that if such monitoring is done at a lower level, where the results are immediately written to a file (or files, one per processor), instead of being kept in memory, the overhead will be acceptable, even with the naïve monitoring approach.

For a more in-depth test of our system, we tried the system on non-deterministic programs. For this, we converted a program which solves the travelling salesman problem from Parlog to KLIC. This program was originally written to perform experiments with speculative parallelism in Parlog [4]. Speculative parallelism results from solving the problem with a number of low-priority solver processes. For our experiments here, a fixed number of equal-priority solver processes was created. If more than one solver process is provided, then the program can behave non-deterministically.

problem	ex. time	naive	non-ann.	ann.
6	0.027±0.005	18663	730	117
10	0.143±0.012	110425	2386	309
15	0.497±0.029	414500	4496	453
20	4.472±0.174	3289836	37348	2917
25	8.266±0.090	7383687	57576	3485
30	69.316±0.667	>60000000	237748	14901

Figure 7: Measurements for Travelling Salesman Problem

The measurements we made for the KLIC program are summarized in Figure 7. Each row of the table represents the results for one instance of the travelling salesman problem. The first column identifies the problem: the number given is the number of nodes that the salesman must visit, with the route cost matrix generated randomly. All the results are obtained by using 2 processes. The second column gives the average execution times in seconds for each problem (without monitoring), with the standard deviations. Problems 6, 10

[1] In our actual system, we do generate a 'stub' so that a one-node trace tree is generated.

and 15 are averages of 3 timings, while 20, 25 and 30 are averages of 5 timings. The third column is the size of the trace tree (in number of characters) generated by the naïve record version of the program. The size for problem 30 is an estimate, as we were unable to store such a large trace tree. The fourth column gives the trace tree size using automatic determinism analysis, without user annotations. The last column gives the trace tree size with user-annotated locally deterministic goals. The redundant node optimization is also implemented in the last two columns' measurements. All the measurements are performed on a SparcStation 10/41.

Currently, we are using a very simple scheme to uncover determinism, and the scheme can be easily improved. In fact, for the predicates which the system was not able to determine to be locally deterministic, but which were locally deterministic upon inspection (and labelled as such by user annotation), the Andorra-I determinism analyser was able to find that these predicates are deterministic.[2] There is therefore good reason to expect that a more sophisticated analyser can detect all the determinism in this program without aid from the user. Nevertheless, both with and without user annotations, the size of the tree is much smaller than the naïve version.

If the problem is run with 1 process only, then it is deterministic, and the versions with determinism analysis and annotations effectively do not generate a trace tree, whereas the naïve version would still generate trace trees of similar sizes to those it generated for 2 processes.

The Parlog version of the program performed over 2.1 million reductions for problem 25, and as the KLIC version of the program is quite similar, we expect it to have performed a similar amount of procedure calls. The size of the trace tree, with determinism analysis, is quite small for these problems. In fact, our results suggest that the size of the trace tree increases at a slower rate than the increase in execution time for larger problems.

problem	original	record
25	8.266±0.090	8.354±0.154
30	69.316±0.667	69.572±0.880

Figure 8: Comparison of execution time for unmonitored and record versions of the travelling salesman program

We have not shown the execution time for the record version of the program in Figure 7, because we were not able to obtain any meaningful differences in execution time between the original and record versions of the program for problems smaller than 25. Even for problems 25 and 30, the differences are well within the margin of error, with the difference being less than 0.5% for problem 30. In fact, other factors, such as what other processes were executing on the machine, seem to have a much greater impact on the execution time than the overhead of recording. Thus, we believe our method is highly suitable for monitoring the execution of KLIC programs, as many KLIC programs exhibit a high degree of determinism.

However, there can always be atypical programs that contain a lot of non-determinism. In such cases, recording the execution will impose a higher overhead. The naïve version simulates the worst case situation where every single goal is not totally deterministic. Even in such cases, our data suggest that recording is still possible for programs that do not run for very long. We also believe that a lower level monitoring system will be able to cope with such programs with very little overheads.

As we do not have a parallel version of KLIC, we cannot directly verify that our scheme does reproduce the same behaviour at replay as when the program was recorded, because the execution of sequential KLIC program is deterministic. We have tested the same program conversion technique with the JAM parallel Parlog version of the travelling salesman problem on a Sequent Symmetry, and the replay version does duplicate the results of the record version. We were able to verify that KLIC version in an indirect fashion: altering the priorities of the goals in the replay version. Without the constraint of the trace tree, the altered priority changes the results. However, our experiment showed that, as expected, the record and replay versions produced the same result despite the different priorities on some of the critical goals. This strongly suggests that the trace tree does control the replay execution as we expected.

[2]The predicates have to be slightly modified so that they conform to being legal Prolog predicates without cuts.

5 Conclusions and further work

The Instant Replay approach to debugging seems to be a feasible way of providing repeatable execution for KLIC programs at a very low cost. Repeatable execution solves a major problem with debugging concurrent logic programs that are actually executed in parallel, and is useful for most, if not all, debugging approaches for concurrent logic programming languages. It is also useful for performance monitoring and performance debugging. We believe that the method we used is very general, and can be easily applied to other concurrent logic programming languages.

Further work includes improving the determinism analyser so that the user would need to annotate as few predicates as possible. It would also be useful to integrate the monitoring seamlessly into the KLIC environment so that the user actually sees the original program being debugged, instead of the converted version, which is much less humanly readable. It may also be interesting to investigate implementing the method at a lower level than a source-to-source transformation, as such an implementation would be even more efficient, especially if the amount of monitoring that needs to be done is large.

6 Acknowledgements

We gratefully acknowledge the support of ICOT for this work. We would also like to thank Vítor Santos Costa, John Gallagher, Manuel Hermenegildo, Manuel Carro, Enrico Pontelli for helpful discussions on abstract interpretation. We would also like to thank John Lloyd for discussion on declarative and algorithmic debugging. We thank Takashi Chikayama for many general discussions on our system.

References

[1] M. Brayshaw. An Architecture for Visualizing the Execution of Parallel Logic Programs. In *International Joint Conference on Artificial Intelligence*, 1991.

[2] T. Conlon and S. Gregory. Debugging Tools for Concurrent Logic Programming. *The Computer Journal*, 35(2):157–169, 1992.

[3] M. Ducassé. Abstract Views of Prolog Executions in Opium. In V. Saraswat and K. Ueda, editors, *Logic Programming: Proceedings of 1991 International Symposium*, pages 18–32. The MIT Press, 1991.

[4] S. Gregory. Experiments with Speculative Parallelism in Prolog. In D. Miller, editor, *Logic Programming: Proceedings of the 1993 International Symposium*, pages 370–388. The MIT Press, 1993.

[5] R. A. Kowalski. *Logic for Problem Solving*. Artificial Intelligence Series. North Holland, 1979.

[6] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Transactions on Computers*, Apr. 1987.

[7] V. Santos Costa. *Compile-time Analysis for the Parallel Execution of Logic Programs in Andorra-I*. PhD thesis, University of Bristol, 1993.

[8] E. Shapiro. *Algorithmic Program Debugging*. The MIT Press, 1983.

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

[10] J. Tatemura, H. Koike, and H. Tanaka. HyperDEBU: A Multiwindow Debugger for Parallel Logic Programs. Technical report, University of Tokyo, 1992.

[11] E. Tick. *Parallel Logic Programming*. The MIT Press, 1991.

Visual Programming System for Parallel Logic Languages

Jiro Tanaka

Institute of Information Sciences and Electronics,
University of Tsukuba, Tsukuba, Ibaraki 305, Japan
jiro@is.tsukuba.ac.jp

Abstract

Though parallel logic languages have lots of good features, the readability of those languages are not good enough. Therefore, we have developed visual programming system PP for parallel logic language GHC. Each definition clause of GHC program is expressed as a figure along with its textual representation. The system graphically assists the program input and its execution. The system has actually been implemented in SICStus Prolog and available on conventional Unix workstations. Examples of actual program input and its graphical execution are shown.

1 Introduction

Various kinds of parallel logic languages, i.e., PARLOG [Clark 86], Concurrent Prolog [Shapiro 83] and Guarded Horn clauses (GHC) [Ueda 85], have been proposed [Shapiro 89]. The basic computation mechanisms of these languages are quite similar. Horn clauses with guards are used for defining predicates, goals are executed in parallel, and they have some synchronization mechanisms between goals.

Though these languages have lots of good features, such as *expressiveness*, language *simplicity* and *efficiency*, it is often said that the readability of those language is low. For beginners, they often have *mental* difficulty to learn. For even the skilled programmers, parallel logic languages are rather difficult languages to read.

Therefore, we have developed visual programming system PP [1] for those languages. It enables us to input program by figures. It can also *graphically* execute the program. As a target language for our system, we chose a flat subset of GHC which allows only system predicates in guard [2].

1.1 Low readability of GHC

Our observations on the low readability of GHC program and the merits of graphical representation can be summarized as follows.

- **Arrangement of goals** Goals in each definition clause are shown in an one-dimensional manner. It makes difficult to understand the relations between goals. Therefore, if we can position goals *graphically*, it leads to an easier understanding of a program.

[1]PP stands for "Program-less Programming" or "Pictorial Programming."

[2]Though we picked GHC as our target language, it does not mean our system is only applicable to GHC. Our system is also adaptable to other parallel logic languages or Prolog with slight modifications.

- **Avoiding names of arguments** Each goal exchanges information through its arguments. Arguments of goals can be shared by *sharing* their names, which is often not easy to trace. It will be convenient, if this *sharing* relation is expressed *graphically*.
- **Argument mode** Similar to Prolog, GHC does not describe *explicitly* whether an argument is used as an input or an output. However, we usually know the *intended* usage of an argument. It will make GHC programs more readable, if we can express the input and output mode of variables explicitly.
- **Integrated environment** We do not have the good module structures in GHC. It makes the understanding of the program difficult. Therefore, we would like to provide an *integrated* visual environment, which dynamically manages the definition clauses of GHC programs.

1.2 Problems on existing visual programming system

Normally the effort for visual programming starts from visualizing existing text based programming languages. Futamura's PAD is one of such effort [Futamura 81]. However, the visualization of an existing conventional language is not very interesting. It is because the language level of the existing conventional language is too low. We cannot use the same framework for visualizing program execution. Side-effects of conventional languages are also harmful for visualization.

Since declarative languages are side-effect free, these languages are more suited for visualization. Various visualization proposals have already been done for functional languages [Dennis 74, Davis 81, Keller 81], an object-oriented language [TGS 89], logic programming languages [Kurita 84, Cox 85] and parallel logic programming languages [Ringwood 89, Kahn 92]. Other approach is the proposal of iconic-based visual programming language, such as HI-VISUAL [Hirakawa 87].

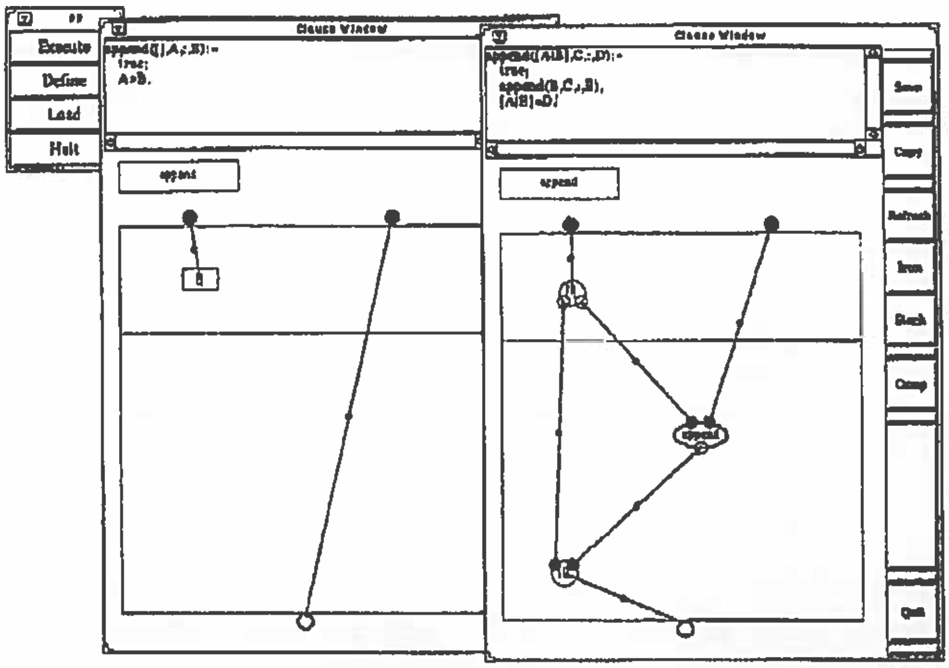
However, most of these efforts do not seem to be successful enough. Some of them were just proposals. Though experimental implementations were done for the rest of proposals, most of them could not survive [3].

Usually visual programming system is *experimentally* developed. The limited numbers of primitives are actually implemented. It is only good for expressing *toy* programs. The number of users, workable machines and environment are also very limited. Therefore, visual programming system is never used for actual software development.

We analyzed the reason that main difficulty arises from proposing new visual programming languages, completely separated from the text-based language. Generally speaking, program language should be *judged* by the total number of built-in functions, libraries, matching to Unix environment and available software tools.

We adopted a *hybrid* approach. In our system, graphical representation is always displayed accompanied with its textual representation, i.e., there always exists a textual representation which corresponds to the graphical representation. Since a textual representation is just a GHC program, it is always possible to utilize the latest implementation of GHC. It is also possible to use our system for part of the total system, mixed with the ordinary GHC system. We believe that our approach is more realistic and promising approach.

[3]The exceptions are Prograph [TGS 89], Pictorial Janus [Kahn 92] and HI-VISUAL [Hirakawa 87].



```
append([],A,B):-
  true;
  A=B.
```

```
append([A|B],C,D):-
  true|
  append(B,C,E),
  [A|E]=D.
```

Figure 1: Program input example of *append*

2 Visual programming system for GHC

In PP, as mentioned above, each definition clause of GHC program is expressed as a figure along with its textual representation. A program input example of *append* is shown in Figure 1.

When we start the system, PP menu pops up initially. PP menu consists of four buttons, i.e., "Execute," "Define," "Load" and "Halt." "Execute" is used to execute a program, after finishing the input of the program. "Define" is used to define a definition clause initially. "Load" is used to load definition clauses from the store. "Halt" is used to exit PP.

We can see two Clause Windows in Figure 1. These two windows correspond to the two definition clauses of *append*. Each Clause Window consists of two sub-windows, i.e., textual-view window and graphical-view window. The former displays the textual representation of a program clause. The latter displays the graphical representation of the same program clause.

In PP, a new Clause Window is created, if we push "Define" button of PP menu. Two sub-windows of the Clause Window, i.e., textual-view window and graphical-view window are initially empty. We can input a clause definition from either sub-windows. If we input a figure from the graphical-view window, the corresponding textual code is automatically generated by the system. On the other hand, if we input the text from the textual-view window, the corresponding figure is also automatically created. The change of one sub-window immediately effects the display of the other sub-window, i.e., two sub-windows are *causally connected* to each other.

PP can be looked at from the viewpoint of MVC (Model-View-Controller), which is originally proposed in Smalltalk-80 [Cox 86]. Model corresponds to the store which contains definition clauses of the program. PP provides two Views, i.e., textual-view and graphical-view, for each definition clause. Controllers are attached to each View and accept messages from the user.

We also tried to minimize the *modes* of the system. For example, PP menu is always available, while PP is running. Since there is no specific execution mode, we can perform various kinds of works *in parallel*. For example, we can open arbitrary numbers of Clause Windows. We can perform *define*, *load* and *execute* in parallel. Since PP is completely fitted in X Window system and Unix, it can be utilized as one of software tools in the process of software development.

3 Input from graphical-view window

Figure 2 shows the program input process when the first definition clause of *append* is entered from the graphical-view window.

Graphical-view windows consists of three regions, i.e., predicate-name, guard and body regions. We can input a predicate name, which is initially set to *no_name*. This is done by clicking the predicate-name region and typing in the new predicate name. We can click the appropriate place of graphical-view window to create input and output arguments. Graphical-view window is clever enough to create appropriate figure elements, such as input arguments, output arguments, goals and connecting lines, depending upon the place where the click from the user is entered.

We can also *graphically* edit the definition clause. For example, we can *delete* or *move* goals *graphically*. When we move goals, arguments of goals and connecting arcs are also moved automatically.

The input from graphical-view window also effects the textual-view window. The corresponding textual codes have been created *incessantly*, as we input an element of a figure from the graphical-view window.

The textual notation is the one of ordinary GHC, except that ";" is used for expressing guard [4] and ":" is used to show the mode of arguments. We arrange input arguments before ":" and output arguments after ":" [5]. As mentioned before, GHC has no *explicit* notion of input and output. Therefore, the distinction of input and output is just for convenience for the graphical display. We can simply neglect the mode, if needed, at run time.

We should also note that we did not use any name of variable in the graphical-view, window so far. Variables which consists of one capital letter, such as "A" and "B" are automatically generated.

One problem of generating textual code from the graphical notation is that sometimes textual notation cannot be determined uniquely. For example, we have generated the following text in the case of Figure 2.

```
append([A|B],C,:,D):-
  true;
  append(B,C,:,E),
  [A|E]=D.
```

However, some people may not be satisfied with the above code and prefer the following code.

[4]This comes from the implementation restriction of SICStus Prolog.

[5]This notation is not *upward* compatible to existing GHC programs. To make it compatible, we also need to prepare *annotations* to specify the mode of arguments.

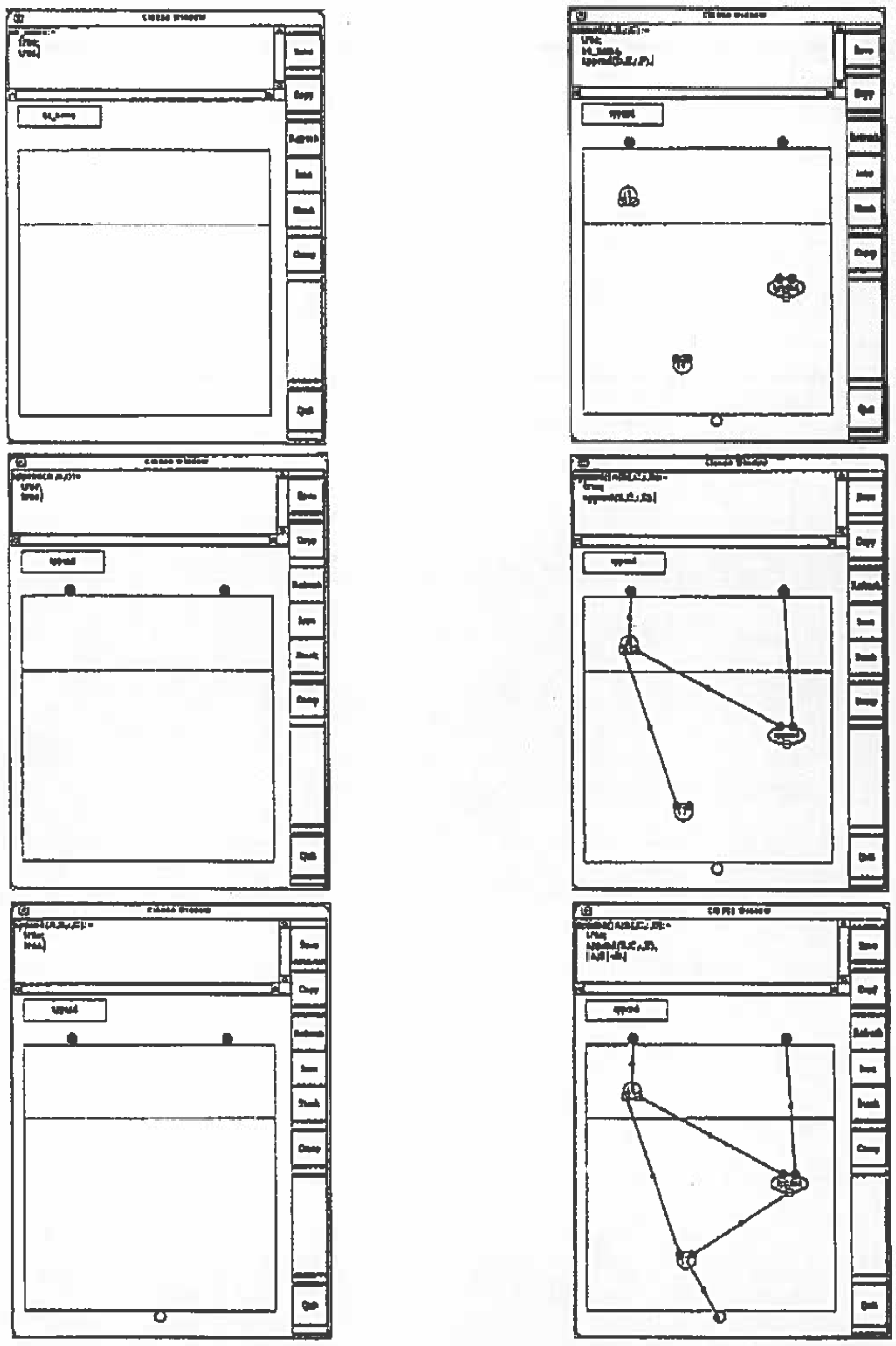


Figure 2: Input from graphical-view window

```
append(A,B,:,C):-
  A=[D|E];
  append(E,B,:,F),
  [D|F]=C.
```

Of course, this is the matter of preferences in a certain extent. Our design choice is that we substitute guard unification to head, if possible, and guard or body is displayed as *true*, even if it is empty.

4 Input from textual-view window

Figure 3 shows the case, in which the same definition clause of *append* is entered from the textual-view window. The textual-view window is just an Emacs window and we can input text from this window.

The input from textual-view window effects the graphical-view window incessantly. Every time we input delimit symbols, such as "(", ",", and":-", drawing routine is triggered and the corresponding graphical view is created and displayed.

As you see in Figure 3, PP has a very *sophisticated* parser which *completes* the insufficient information from textual-view and always generates a syntactically correct GHC definition clause.

5 Labeled variables

One of the advanced feature of PP is the treatment of variables. We did not use the name of variable in the graphical-view so far. In the textual-view, we simply used variables which consists of one capital letter.

However, we sometimes want to call a certain variable with a name. In general, the name of the variable may include the information about the *usage* of the variable. For example, a pair of variables which are used to express a difference list may be called "Head" and "Tail."

We prepared *labeled* variables for such purposes. PP has two kinds of variables, i.e., annamed and labeled variables. An unnamed variable consists of upper-case letters and numbers only. They do not include lower-case letters in their name. They are not displayed in the graphical-view.

On the other hand, labeled variables include lower-case letters in their name, e.g., "Head" and "Tail". Labeled variables are displayed in the graphical-view.

Figure 4 shows the example in which labeled variables are included in the graphical-view. In PP, a duplicate use of the label is allowed in the graphical-view, which is distinguished by putting numbers at the postfix position in the textual-view.

Figure 4 may need some explanation. The predicate *append* has two input arguments and one output argument. We labeled those as *FList*, *SList* and *Applist* in the graphical-view. Since *append* is called recursively, it is informative to use the same labels also in the recursive call part. In PP, identical labels are distinguished by putting numbers at the postfix position in the textual-view. However, as mentioned before, guard unification is substituted to head, if possible, in the textual-view. Therefore, only *FList*1 shows up in the textual-view.

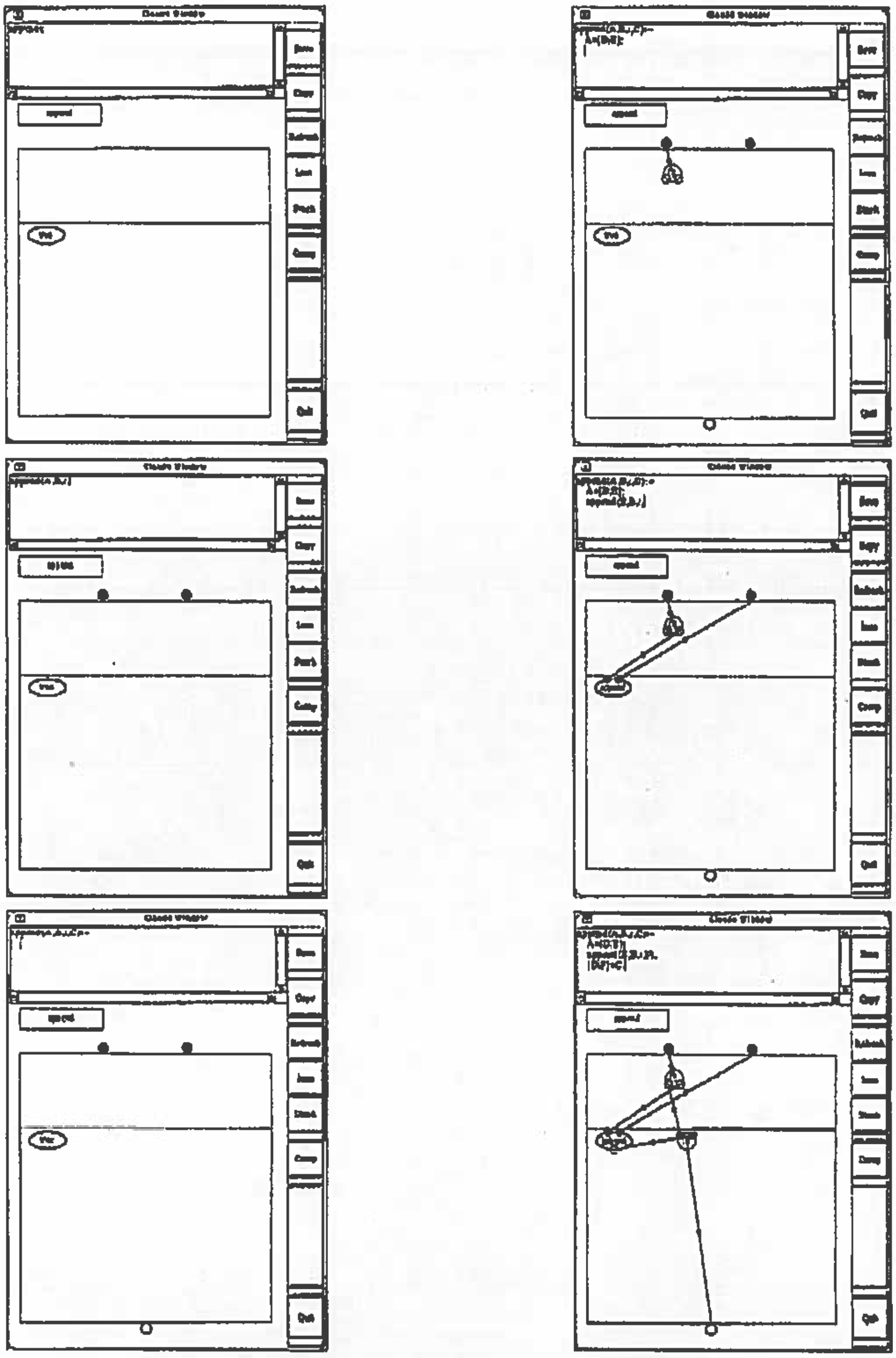
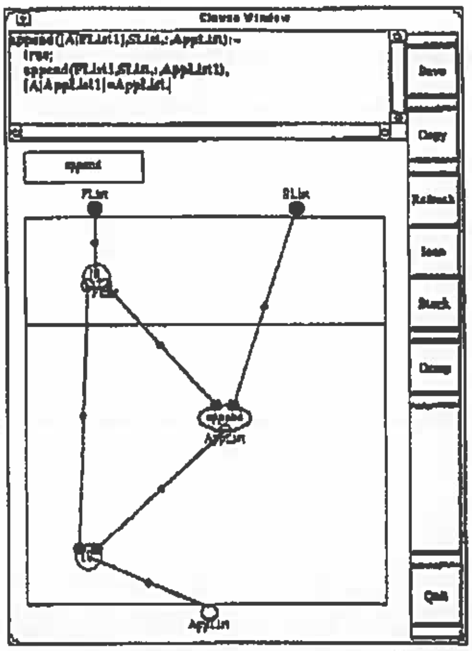


Figure 3: Input from textual-view window



```
append([A|FList1],SList,AppList):-
    true;
    append(FList1,SList,AppList1),
    [A|AppList1]=AppList.
```

Figure 4: An example of labeled variables

6 Program execution example

Figure 5 shows the *demo_app* example. In this case, the definition clause of *demo_app* was entered from the textual-view window. Note that *pretty drawing*, which is compatible to the textual notation of list, has been introduced to increase the readability of the graphical representation.

This *demo_app* program just tries to append two lists. We can execute this program by pushing "Execute" button of PP menu. Execution Window is created, if we push "Execute" button.

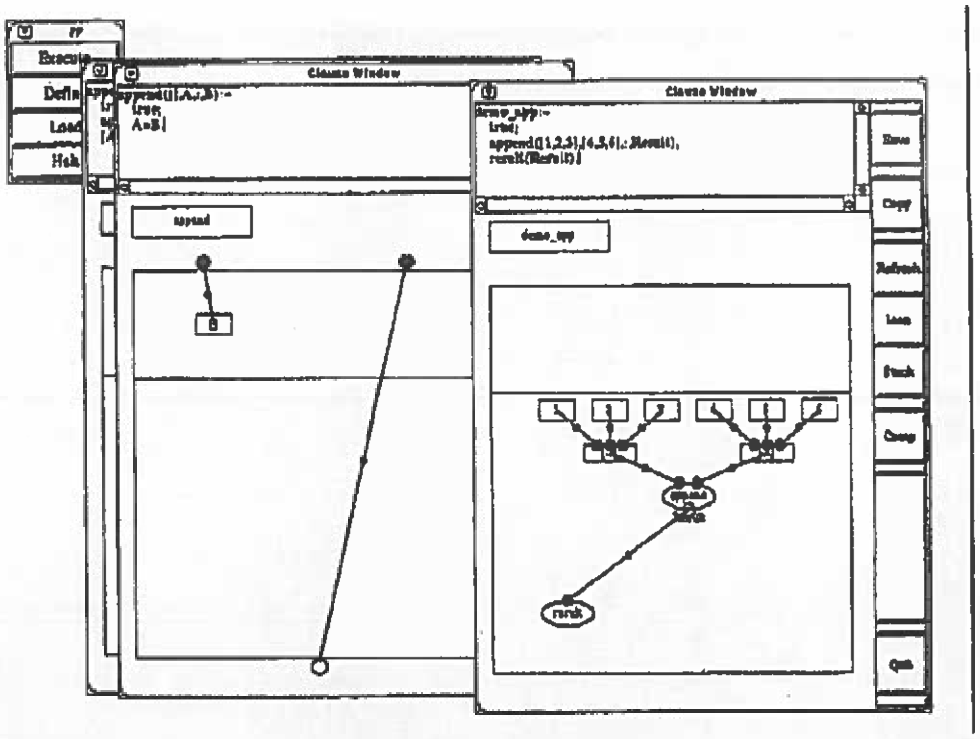
We can execute GHC goals in "Breadth-first," "Depth-first" or "Parallel" manner [6].

We enter a GHC goal, in this case, *demo_app* goal, to Execution Window and push "Run" or "Step" button of Execution Window [7]. The snapshots of program execution are shown in Figures 6 and 7.

In GHC, the end of computation means that all goals are disappeared from the goal queue. the goal which will be reduced to itself. Note that the "result" goal is used to keep the computation result on Execution window [8].

The execution by Execution Window is quite interactive. It is possible to edit goals in the middle of the execution. For example, we can delete goals, add new goals, or modify the connection of arguments in the middle of goal execution.

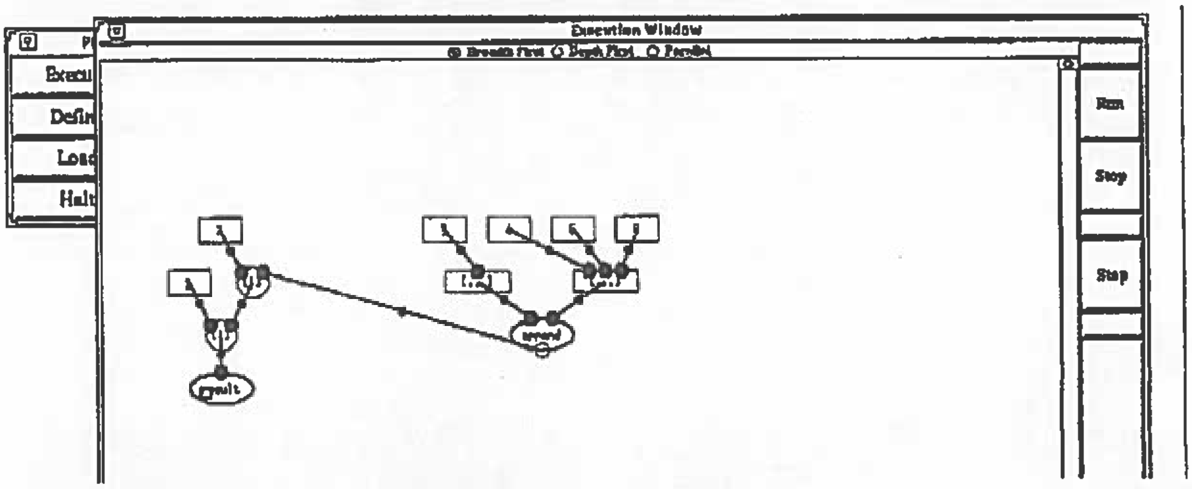
[6] "Parallel" means to reduce all GHC goals in a goal queue at the same time.
[7] "Step" means stepwise execution.
[8] The "result" goal can be defined as a predicate which calls itself recursively.



```
append([],A,B):-
    true;
    A=B|
```

```
demo_app:-
    true;
    append([1,2,3],[4,5,6],Result),
    result(Result)|
```

Figure 5: *demo_app* example



● Breadth First ○ Depth First ○ Parallel

Figure 6: Program execution snapshot(1)

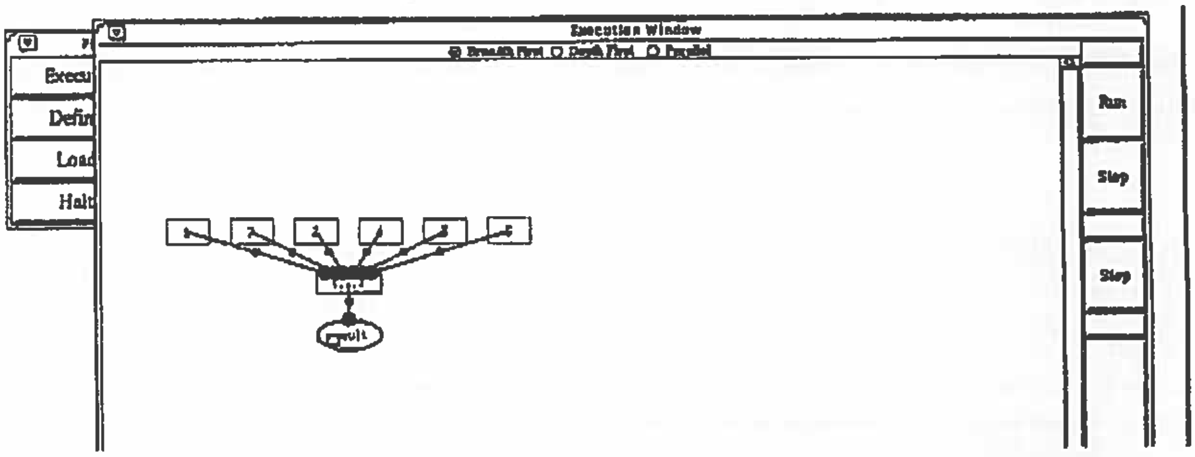


Figure 7: Program execution snapshot(2)

7 System implementation history

System implementation effort of PP has been started in 1989. The first version and the second version have been implemented 1990 and 1992, respectively. Both systems were developed on top of PSI-II. The first version is written in GHC, the second, in KL1. These was carried out as a part of the Fifth Generation Computer Project. The second version was demonstrated at the FGCS'92 in 1992. This version has been registered as ICOT Free Software.

The system we presented in this paper is the third version, which was started 1992, when the author was working at Fujitsu Laboratories. The third version was developed on top of ordinary Unix workstations. SICStus prolog was used as an implementation programming language. Graphic Manager (GM) package of SICStus Prolog was used for graphics and interaction in the X Window environment. (GM is written using the InterViews system.)

8 Concluding remarks

We have developed visual programming system PP, which graphically assists the input of a GHC program and its execution. In PP, definition clauses of a GHC program, which are often difficult to understand, can be expressed as a figure along with its textual representation. Main features of PP can be summarized as follows.

- **Graphical input** We can input definition clauses of GHC program by figures. Textual form of the definition clause is automatically generated by the system.
- **Textual input** Clause definitions are also entered by text. In such case, graphical representation is automatically created. Textual and graphical forms are *causally connected* each other.
- **Editing graphical input** We can *graphically* edit the definition clause at any time.

- **Save and load** Textual and graphical forms can be saved to the store and can be loaded, if necessary.
- **Integrated environment** The system provides an *integrated* visual environment, which dynamically manages the definition clauses of GHC programs.
- **Graphical execution** We can execute goals graphically and interactively.

The preliminary evaluation of PP shows that the programming work using PP is very comfortable, comparing to the ordinary programming work. The operational feeling of PP is very similar to that of Macintosh. The user can *enjoy* the programming work. PP is really interactive and we can start using without sufficient preliminary knowledge of GHC and PP.

In the case of the visualization of conventional languages, we often need to insert primitives for dynamic visualization to the source code. However, PP is a fully automatic visual system and we do not need to insert such primitives. it is because the gap between the graphical representation of the *static* program and that of program execution is small in GHC.

Since the idea of expressing a program graphically is not a completely new idea, there are lots of related works as discussed in Section 1.2. Comparing to these works, the critical difference exists in the adoption of the *hybrid* approach. In PP, graphical representation can naturally matches to the textual representation. Therefore, we can utilize the latest implementation of the textual world. It is also possible to use PP for part of the total system, mixed with the text based system.

The further refinement up of the system and adding animating capability such as seen in Pictorial Janus [Kahn 92] are the topics for our future works.

9 Acknowledgments

The authors would like to express thanks to Yukiko Ohta, Simon Martin and Katsujiro Nakano for their help and programming support.

References

[Clark 86] K. Clark and S. Gregory: PARLOG: Parallel Programming in Logic, ACM Transactions on Programming Languages and Systems, Vol.8, No.1, 1986, pp.1-49.

[Cox 85] P.T. Cox and T. Pietrzykowski: Lograph: A Graphical Logic Programming Language, in Proceedings of IEEE COMPINT 85, Montreal, 1985, pp.145-151.

[Cox 86] B.J. Cox: *Object-Oriented Programming: An Evolutionary Approach*, Addison-Wesley, 1986.

[Davis 81] A.L. Davis and S.A. Lowder: A Sample Management Application Program in a Graphical Data-Driven Programming Language, in Digest of Papers Compcon Spring 81, February 1981, pp. 162-167.

[Dennis 74] J.B. Dennis: First Version of a Data Flow Procedure Language, Lecture Notes in Computer Science, Vol.19, 1974, pp.362-376.

[Futamura 81] Y. Futamura, T. Kawai, H. Horikoshi and M. Tsutsumi: Development of Computer Programs by PAD (Problem Analysis Diagram), in Proceedings of the Fifth International Software Engineering Conference, IEEE, New York, 1981, pp.325-332.

[Hirakawa 87] M. Hirakawa, S. Iwata, I. Tanimoto, M. Tanaka and T. Ichikawa: HI-VISUAL Iconic Programming, in Proceedings of the 1987 Workshop on Visual Languages, August 1987, pp. 305-314.

[Kahn 92] K.M. Kahn: Concurrent Constraint Programs to Parse and Animate Pictures of Concurrent Constraint Programs, in Proceedings of the International Conference on Fifth Generation Computer Systems 1992, ICOT, June 1992, pp.943-950.

[Keller 81] R.M. Keller: Applications of Feedback in Functional Programming, in Proceedings of 1981 Conference on Functional Programming and Computer Architecture, ACM, 1981, pp.123-130.

[Kurita 84] T. Kurita and K. Tamura: Dialog.I: An Iconic Programming System Based on Logic Programming, Bulletin of the Electrotechnical Laboratory, Vol. 48, No.12, 1984, pp.966-975.

[Ringwood 89] G. Ringwood: Predicates and Pixels, *New Generation Computing*, Vol.7, No.1, 1989, pp.59-80.

[Shapiro 83] E. Shapiro; A Subset of Concurrent Prolog and Its Interpreter, ICOT Technical Report, TR-003, 1983. (Also in *Concurrent Prolog: Collected Papers*, Vol.1, The MIT Press, 1987, pp.27-83.)

[Shapiro 89] E. Shapiro: The family of Concurrent Logic Programming Languages, *ACM Computing Surveys*, Vol.21. No.3, September 1989, pp.413-510.

[TGS 89] The Gunakara Sun Systems: *Prograph Tutorial*, The Gunakara Sun Systems, 1989.

[Ueda 85] K. Ueda: Guarded Horn Clauses, ICOT Technical Report, TR-103, 1985. (Also in *Concurrent Prolog: Collected Papers*, Vol.1, The MIT Press, 1987, pp.140-156.)

Debugging and Performance Debugging Tools for Parallel Logic Programming Language Fleng

Junichi Tatemura, Hanpei Koike, Hidehiko Tanaka
{tatemura,koike,tanaka}@mtl.t.u-tokyo.ac.jp
Faculty of Engineering, The University of Tokyo

1 Introduction

For practical parallel computers to come into wide use, it is important that they execute practical application programs efficiently. One important problem is to develop programming environments for parallel programs. Especially debugging and performance debugging tools are essential to develop efficient, highly parallel programs.

Debugging parallel programs is much more difficult than debugging sequential programs, because many processes run simultaneously and interact with each other.

A parallel program should run fast, because the purpose of a parallel machine is to speed up execution. Therefore, a performance debugger is necessary for a parallel programming system.

We have developed a debugger and a performance debugger for parallel logic programs. In this paper, we describe these tools.

2 Committed-Choice Language Fleng

Committed-Choice Languages (CCLs) such as Guarded Horn Clauses (GHC) ,Concurrent Prolog and PARLOG are parallel logic programming languages which introduce a control primitive "guard" for synchronization. Fleng [1] is a CCL designed in our laboratory. We are developing the Parallel Inference Engine PIE64 [2] which executes Fleng programs. Fleng is a simpler language than other CCLs ; Fleng has no guard goal, and only the head realizes the guard mechanism.

In order to realize communication and synchronization in concurrent logic programs, unification in CCL is divided into two classes : *guarded unification* and *active unification*. Guarded unification is applied in the head part of a clause, and variables in a goal are prevented from being substituted. Such unification is *suspended* until these variables have values. Active unification is applied in the body part of a clause and is able to substitute values for variables of goals. A clause in a Fleng program defines control flows through *goal reduction* and data flows through *active / guarded unification.*

3 Multi-window Debugger HyperDEBU

3.1 Feature of the Debugger

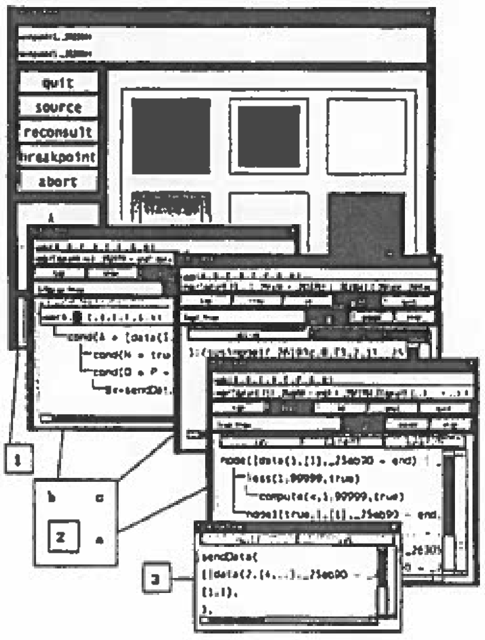
We have developed a multiwindow debugger HyperDEBU which provides a multi-dimensional interface.

A sequential program has only one thread of execution, which can be debugged with a sequential interface. On the other hand, a parallel program has multiple complicated control/data flows which are considered to be multi-dimensional information. If a sequential interface is used to debug a parallel program, the bottleneck between a programmer and the program makes it difficult to examine and to manipulate the execution of the program. Therefore, a multi-dimensional interface is necessary to debug a parallel program.

Since a user compares a model represented by a debugger with the expected behavior of the program when he/she debugs a program, the debugger must provide a view of the kind he/she wants. Accordingly, the debugger must provide views which have flexible levels and aspects of abstraction.

Most conventional multiwindow debuggers use a window as a sequential debugger assigned to one of the processes [3]. However, multi-dimensional information cannot be handled well in this way. HyperDEBU provides windows flexible enough for programmers to examine and manipulate complicated structures composed of multiple control/data flows. Tracing links on a window which displays information of a program execution, a user can get an expected window.

The conventional notion of process for CCL is associated with one goal or one sequence of goals. The process model for our debugger is equivalent not to one goal but to all of its subgoals generated by reduction. Let G be a goal and P be a set of goals which are derived from G. We call P "*the process with respect to* G", and call G "*the topgoal of* P". Since a subgoal can be a topgoal, a process consists of some subprocesses. This hierarchy of processes makes the debugger applicable to highly parallel programs.



```
node([data(1,[1],_25eb70 - end)
  less(1,99999,true)
    compute(<,1,99999,true)
sendData(
[[data(2,[4,...],_25eb90 -
[1,1]),
),
```

Figure 1: Overview of HyperDEBU

HyperDEBU consists of the following windows : (1) toplevel-window, (2) process-windows ((a) TREE view (b) I/O tree view (c) GOAL view), and (3) structure-windows. Figure 1 shows an overview of HyperDEBU.

HyperDEBU has these features which cooperate with each other to aid a user to find a bug.

1. **Various views for bug locating.** Since it is necessary to zoom gradually in on the location of the bug as the breadth of a view is kept properly, a debugger's view is required to have the flexibility of views from global to local. A toplevel-window provides a global view. A user can get a process-window to examine detail of any process displayed on the toplevel-window. A process-window enables examination and manipulation of the process. To locate bugs, a user can get a subprocess as another window from this process. The process-window has three views

of the process. Moreover, HyperDEBU has a structure-window which provides a data-level view.

2. **Visualization.** The global view of the toplevel-window visualizes execution of a Fleng program. This function helps a user to comprehend the global situation of the execution, and makes bug locating more efficient.

3. **Breakpoints for parallel execution.** Since a parallel program has multiple threads of control flow, a new mechanism to control the execution is needed in order to debug a highly parallel program. We extend "breakpoints" as a debugger's knowledge given by a user before the execution of the program. The debugger uses this information to control the execution, visualization and static debugging.

4. **Browsing program code.** To comprehend static information of Fleng programs, HyperDEBU has a prototype of the program code browser. This function helps a user to set breakpoints, make static debugging, and correct a source code.

3.2 Program Visualization on HyperDEBU

3.2.1 Basic Approach

A debugger must deal with a large amount of control flow and data flow information in order to visualize execution of a highly parallel program. A Fleng program can be represented by visualizing *goal reduction* as control flow and *guarded / active unification* as data flow. However, visualizing all goals and data is hard to comprehend.

HyperDEBU visualizes these flows using proper abstraction according to a user's intention which is given as additional information. The debugger provides low level abstraction for debugging when it has no information from a user, and, as it is given information, enables the user to debug the program at higher level abstraction.

We introduce "breakpoints" as information which represents a user's intention or points of view. Breakpoints are specified as pairs of "point" and "direction". HyperDEBU has two directions for program visualization: "process" to visualize a process with respect to a goal, and "stream" to visualize a data as a stream.

3.2.2 Visualizing Control Flows

HyperDEBU visualizes creations and state transition of processes. Since a process is defined as a set of goals derived from a goal, each goal in the execution history has corresponding process. HyperDEBU visualizes only processes with respect to some particular goals. Each process is displayed as a rectangle. A color of the rectangle indicates the state of the process. A nest of rectangles indicates the relation between a process and its subprocess. A topgoal of a process is displayed when the mouse cursor enters the corresponding rectangle. Clicking a rectangle generates a new process-window for this process.

Contents of all the windows are updated reflecting the state of the execution dynamically. By observing creations and state transition of processes, and by observing modification of data in the arguments of topgoals, a user can comprehend the execution of Fleng program correctly and easily.

3.2.3 Visualizing Data Flows

HyperDEBU visualizes particular *streams* which make main global data flows of the computation. The stream-based communication is a essential programming technique in CCL which enables a continual communication between processes. It is realized using shared variables as follows.

1. One of goals which have a shared variable writes some data structure into this variable (stream output).

2. When the value of the variable is bound, the other goals read it (stream input).
3. The data structure includes new variables which are shared by goals and used to continue the communication.

To represent the stream-based communication, our debugger visualizes (1) communicating goals, (2) shared variables for the communication, (3) outputted data, and (4) inputted data (guard) as Figure 2.

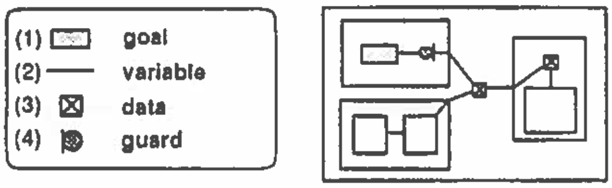
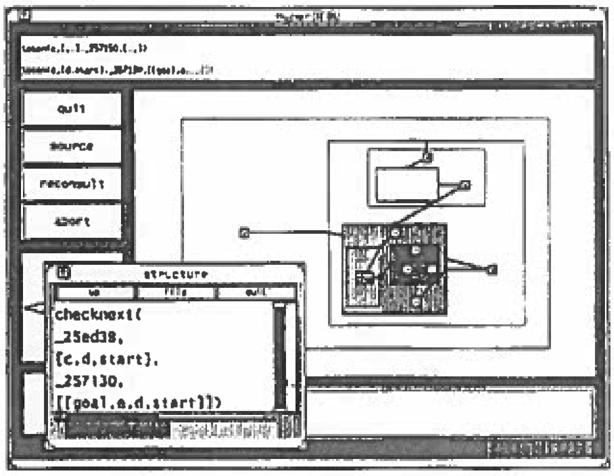


Figure 2: Visualization of Streams

A user specifys an argument of a predicate as a "stream" breakpoint.

3.3 Example

In this section, we will show an example which is a program to solve "good-path problem". This program searches the paths on the directed graph and finds all paths from **`start`** to **`goal`**. To get the solution, **`token`** goals spawn themselves and search the paths from **`start`** for **`goal`**. A **`token`**, which has a node in its argument, spawns a goal **`checknext`** if the node is not **`goal`**. The **`checknext`** checks the definition of the directed graph and spawns a **`token`** for each nodes next to the node if the path from **`start`** to the new node has no loop. If a **`token`** reaches the node **`goal`**, it links the solution with shared variables to make the list of the solutions. However, the erroneous definition of the directed graph makes this program suspend illegally without returning the solutions.



```
checknext(
_25ed38,
[c,d,start],
_257130,
[[goal,a,d,start]])
```

Figure 3: Visualizing the Erroneous Program

First, breakpoints for visualization are placed in order to understand the global situation of the execution. The main control flows of this program are composed of creations of **`token`** goals, and the main data flows are considered as the stream which links all the solutions with a list. So a "process" breakpoint is specified at **token**, and "stream" breakpoints are specified at its arguments. Figure 3 shows the visualization of control and data flows of the program. The nested rectangles represent the processes with respect to **token** goals. These process are linked with a stream in sequence. They searches for solutions in parallel, and when a process finds a solution, it is linked with the stream. However, a process represented as a gray rectangle is suspended and a goal is still linked with the stream in the rectangle. A structure window is opened by clicking this goal, and shows the goal **checknext** is suspended waiting for the data of its argument. This goal stops data flow of the list of solutions. To examine details, a process window can be opened from the gray rectangle in which **checknext** is suspended.

4 Performance Debugger

4.1 Method of Performance Debugging

In this paper, we divide tasks of improving performance of a program into two classes : *performance debugging* to improve inherent parallelism of a program, and *performance tuning* to make partitioning / scheduling of the program suitable for a particular environment. Our approach to develop high-performance parallel programs has two stages :

- description of parallelism with CCL
- mapping processes and data on an actual machine

Although these stages influence each other, dealing with them separately makes the problem of performance clear. On the first stage, a performance debugger is used to modify an algorithm of a program. On the second stage, partitioning / scheduling of the program is optimized using a performance tuner [4], an optimizing compiler, and a mechanism of optimum execution on a run-time system.

We have developed a performance debugger to be used on the first stage. The performance debugger helps programmers to find unexpected sequentiality and to improve parallelism of a program.

4.2 Performance Debugging Tools

4.2.1 Requirements of Functions

Since information on execution of a parallel program is enormous and complicated, it is necessary to comprehend performance of a program as the first task of performance debugging. A performance debugger is required to show the information on the performance summarily, and to help a programmer to extract a part of the program which makes the performance worse.

After extracting the part of the program, the programmer analyzes the relation among control and data flows and finds unnecessary sequentiality which causes the bad performance. A performance debugger needs to support the analysis of sequentiality.

Our performance debugger consists of a performance profiler to comprehend performance of a program, and a dataflow tracer to analyze sequentiality of a program.

4.2.2 Virtual Time Stamps

Our performance debugger deals with inherent parallelism of a program, before tuning partitioning / scheduling of the program. Therefore, we introduce a virtual time stamp to the performance debugger : the time stamp is recorded on an ideal environment where any executable goal is executed immediately.

4.3 Performance Profiler

The performance profiler shows the information on performance of a program summarily. It displays time - parallelism diagrams for any part of the execution of the program.

Figure 4 is a display of the performance profiler. The upper part of the window shows time - parallelism diagrams, and the lower part shows a computation tree diagram which represents control flow of the execution. The horizontal axis of each diagram represents virtual time. When a node of the tree diagram is clicked by a mouse, the corresponding goal is selected and displayed on the window. Then, if "select" button is pushed down, a time-parallelism diagram is displayed for the process with respect to this goal. If "dftrace" button is pushed down, a dataflow tracer for the process is opened.

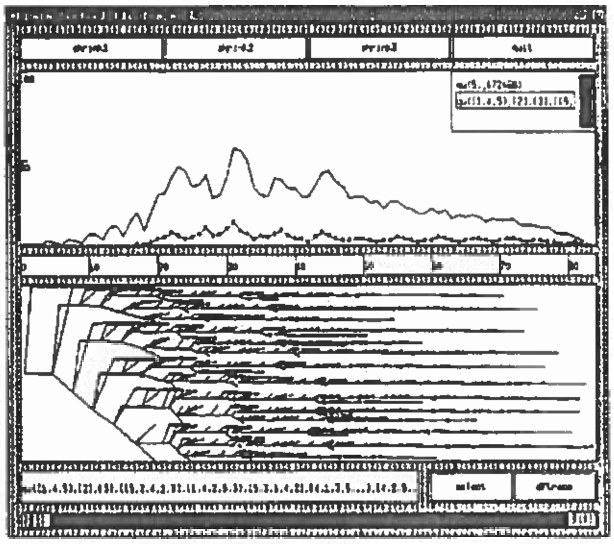


Figure 4: Performance Profiler

Using the performance profiler, a programmer extracts a part of the program which makes the performance worse, and opens a dataflow tracer to analize this part.

4.4 Dataflow Tracer

A dataflow tracer shows graphs composed of control and data flows derived from an execution history of a program. A programmer examines them and removes unnecessary sequentiality by modifying the program.

4.4.1 Input / Output Causality

Execution of CCL program is represented as a history of inputs and outputs and their causality. Since this input / output causality is regarded as a declarative semantics of a CCL program [5], this can not be modified automatically by an optimizing compiler. The dataflow tracer deals with this input / output causality.

Unifications are divided into the following categories to represent inputs and outputs:

- in(Variable, Term) : guarded unification (input)
- out(Variable, Term) : active unification of a variable and data (output)
- unify(Variable_1, Variable_2) : active unification of two variables (output)

The input / output causality of a goal G is represented as a set:

$$IO(G) = \{io_i | io_i = (\{I_{i1}, \cdots, I_{in}\} \rightarrow O_i)\},$$

where O_i is one of outputs and $I_{i1}, \cdots, I_{in}$ are inputs required by O_i.

Every inputs and outputs have virtual time stamps: a time stamp of an input is time when the corresponding data is read for the first time, and one of an output is time when the corresponding data is written.

4.4.2 User Interface

Visualizing I/O Causality Figure 5 shows a dataflow tracer displaying the input /output causality of a program. The horizontal axis represents virtual time. A tree diagram is displayed with its root (a small rectangle) on the left side, and represents the root goal being reduced into two goals. Large rectangles, displayed as leaves of the tree, represent processes with respect to the corresponding goals.

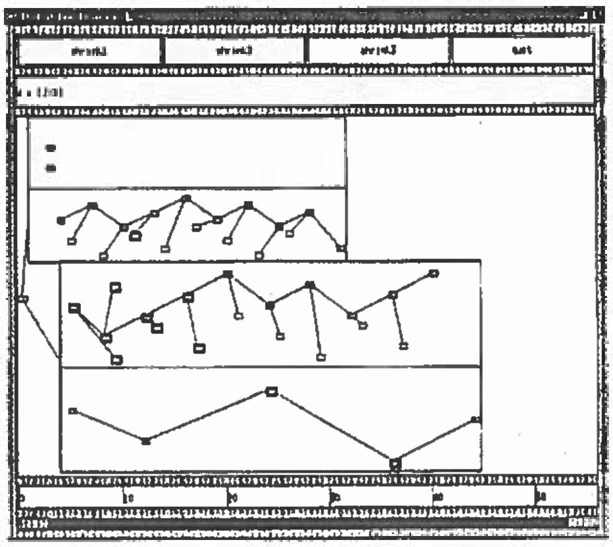


Figure 5: Dataflow Tracer

Execution of a process is represented as a set of inputs and outputs. The upper half of the rectangle displays inputs and the lower half displays outputs. Each input / output is displayed as a small rectangle which position indicates its virtual time. A line connecting two rectangles indicates a pointer from one to another. When a rectangle is pointed with a mouse cursor, the details of the input / output is displayed, and inputs and outputs which have relation to the input / output are highlighted.

A process with respect to a goal can be replaced with a sub-tree which root is the goal and which leaves are processes with respect to its sub-goals.

Tracing Dataflow The dataflow tracer provides a tracing window which is opened when any input / output is clicked. This window is used to trace the following four types of dataflow relationship with an output (input):

- an input/output which has a pointer to the output (input),
- an input/output to which the output (input) has a pointer,

- an input which receives data from the output (an output which sends data to the input),
- an input required by the output (an output which requires the input).

Figure 6 shows the tracing window. The current data is displayed in the center of the window. Four sub-windows display inputs and outputs which have relation to the current data. When a mouse cursor points one of these data, the corresponding rectangle on the process is highlighted. Selecting one of the data, a programmer can traverse the dataflow relationship. The tracks of the traverse is represented as bold lines on the display of the dataflow tracer.

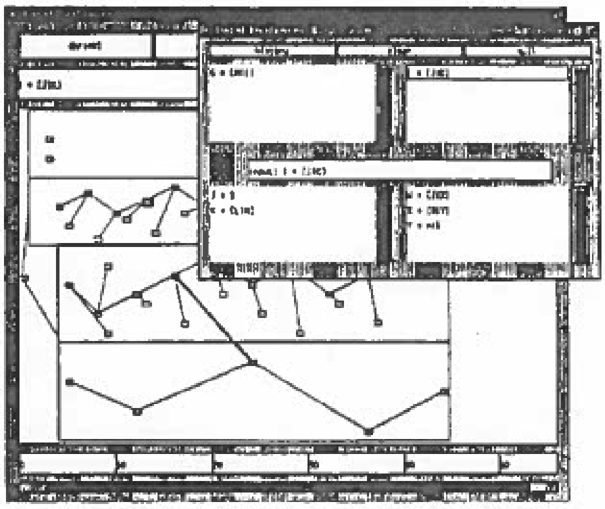


Figure 6: Tracing Dataflow

4.5 Example

4.5.1 Problem for Example

We take up a program to solve " best-path problem" as an example of performance debugging. This program searches the shortest paths from a node to the other nodes on the directed graph. The algorithm to solve this problem is as follows:

> A goal "searcher" traces the graph and searches a path from the node to the other nodes. A searcher on a node creates new searchers for the next nodes. Each searcher calculates cost from the start to the current node, and reports it to a goal "evaluator". The goal "evaluator" manages a record of the shortest paths. When the evaluator receives a report from a searcher, it checks if the cost breaks a record.

A searcher sends a message to the evaluator using stream-based communication. Streams from searchers to the evaluator could be independent of each other. However, the example program has the following performance bug:

> Streams from searchers to the evaluator have sequential relation.

Although this program solves the problem correctly, this sequentiality makes the performance worse.

4.5.2 Scenario of Debugging

In figure 7, a performance profiler displays two time-parallelism diagrams: one is for the whole part of the program and the other is for the evaluator. The diagram for the evaluator is now highlighted (dotted with black points). The parallelism of the whole part is almost always 1. Since sometimes the evaluator is not working, searchers and the evaluator are considered to run by turns.

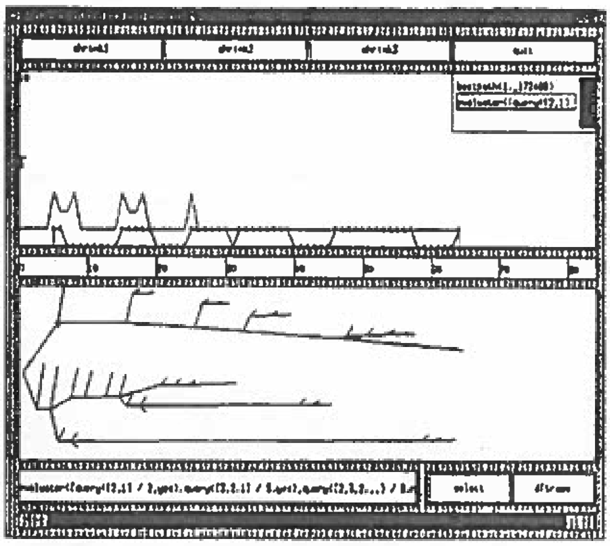


Figure 7: Time-parallelism Diagram (1)

A dataflow tracer is opened to examine the dataflow relationship between the searcher and the evaluator (Figure 8). The upper and lower rectangles on the window indicate the evaluator and the searcher respectively. The relation about queries and answers between their processes is shown as bold lines. When the evaluator receives a message from a searcher, it checks the record of paths and replys to the searcher. Since the evaluator has an interval between tasks of answering queries, the searcher is considerd a bottleneck.

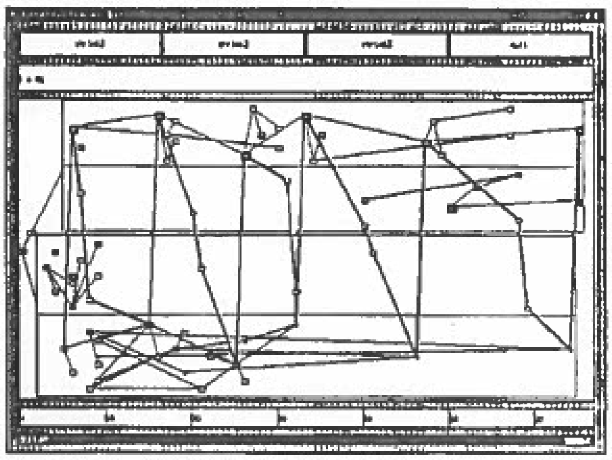


Figure 8: Communication between evaluator and searcher (1)

In Figure 9, the searcher process is replaced by its sub-processes. Two searchers communicate to the evaluator. The lower searcher sends a message on the left side of the rectangle and receive a message on the right side. The interval between them is much longer than the response time of the evaluator. This delay occurs because the message from the searcher is not given to the evaluator before the other searcher completes its communication with the evaluator.

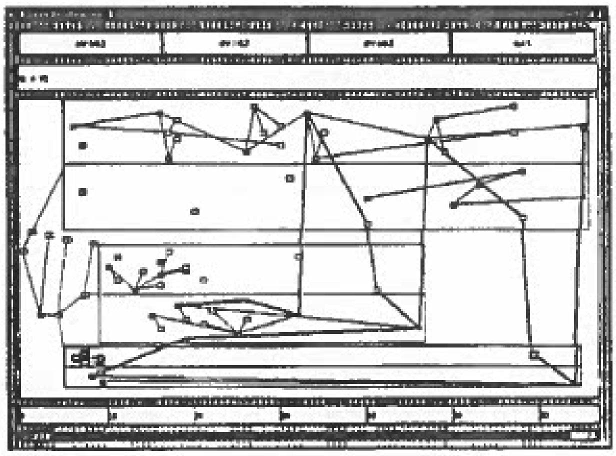


Figure 9: Relation among messages from searchers

A new program is given by removing the performance bug found here. In the program, the streams of the searchers are joined together by "merge". The evaluator has no restriction on the order of receiving messages from two separate searchers.

Figure 10 shows the performance of the new program. The parallelism is enhanced, and the total time is reduced. Figure 11 is a dataflow tracer opened to display the communication between the evaluator and searchers. This indicates that tasks of responding queries are overlapped.

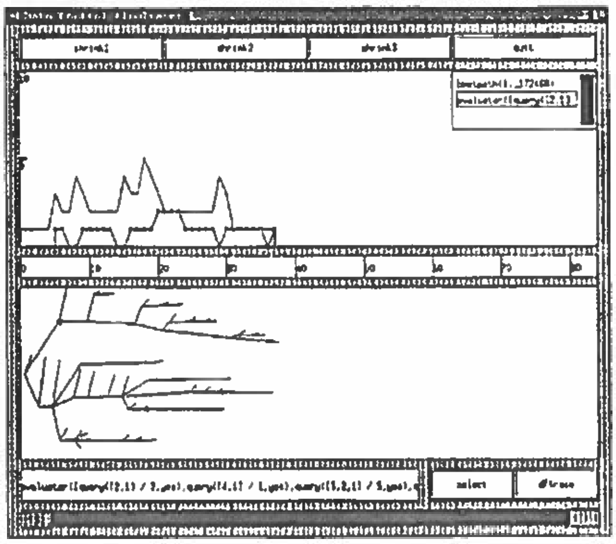


Figure 10: Time-parallelism Disgram (2)

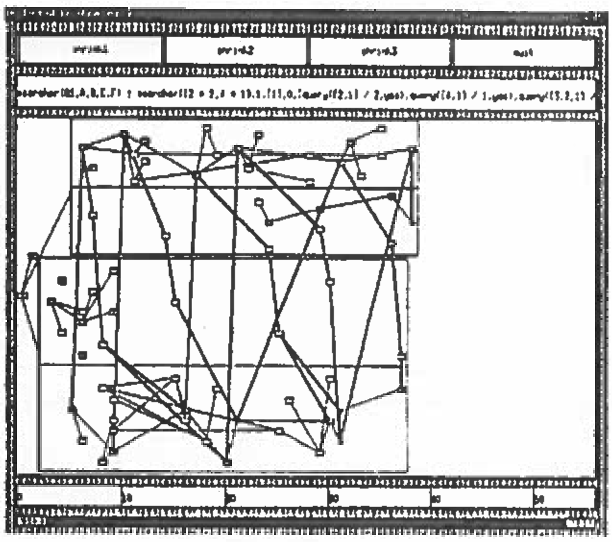


Figure 11: Communication between evaluator and searcher (1)

4.5.3 Results

The programs take the virtual time on the performance debugger and the actual time on PIE64 as follows.

program	virtual time [generation]	actual time [msec]
original	830	282
improved	106	40

The programs solve best paths for 8 nodes, and were executed by the interpreter on PIE64 (using 12 processors). The interpreter made use of the automatic load-balancing mechanism of PIE64, and no static optimization was applyed.

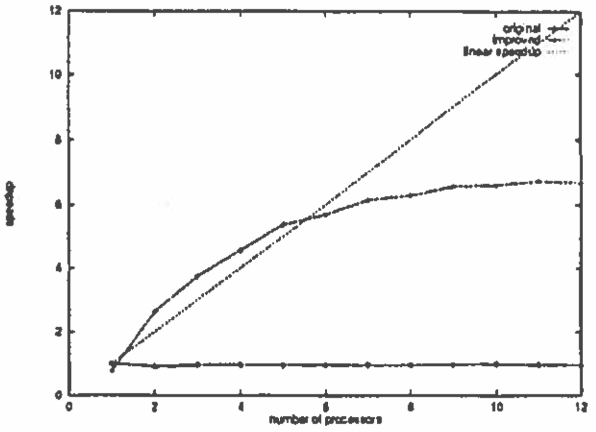


Figure 12: Speedups of the programs

The speedups of the programs are measured as Figure 12. The speedup of the improved program exceeds the linear speedup when the number of processors is small, because the amount of queries from searchers depends on their scheduling. Since the searchers of the original program send queries sequentially and search paths in a depth-first fashion, many records are canceled by following messages.

5 Future Works

We have the following subjects as future works.

5.1 Debugger

- control and analysis of nondeterministic behavior
- reducing the amount of execution history

5.2 Performance Debugger

- visualizing performance data in actual time
- cooperation with an optimizing compiler
- performance debugging an algorithm for which scheduling varies the amount of the computation

6 Conclusion

In this paper, we described a debugger and a performance debugger for Fleng programs. The debugger HyperDEBU provides windows flexible enough for programmers to examine and manipulate a parallel program which has multiple control/data flows. The performance debugger is developed to improve inherent parallelism of a program and helps programmers to find unexpected sequentiality.

References

[1] Nilsson, M. and Tanaka, H.: *Massively Parallel Implementation of Flat GHC on the Connection Machine, Proc. of the Int. Conf. on Fifth Generation Computer Systems*, p1031-1040 (1988).

[2] Koike, H. and Tanaka, H. : *Parallel Inference Engine PIE64*, in *Parallel Computer Architecture*, bit, Vol.21,No.4,1989, pp.488-497 (in Japanese).

[3] Mcdowell,C.E. and Helmbold,D.P.: *Debugging Concurrent Programs*, ACM Computing Surveys, Vol.21 No.4, pp.593-622 (1989).

[4] Aikawa, S., Kamiko, M., Kubo, H., Matsuzawa, F., and Chikayama, T. : Paragraph : A Graphical Tuning Tool for Multiprocessor Systems, International Conference on Fifth Generation Computer Systems 1992, pp. 286-293 (1992).

[5] Murakami, M. : A Declarative Semantics of Flat Guarded Horn Clause for Programs with Perpetual Processes, Theoret. Comp. Sci, Vol.75 No.1/2, pp. 67-83 (1990).

END

CHUKL: Constraint Handling Under KL1

Rong Yang

Department of Computer Science
University of Bristol
Bristol BS8 1TR, U.K.

Abstract

The paper presents a parallel implementation of finite-domain constraint logic programming (CLP(FD)) in the committed choice concurrent logic programming language KL1. To date, both sequential and or-parallel CLP(FD) implementations have been developed, but how to exploit and-parallelism and how to combine both and- and or- parallelism in CLP(FD) have not yet been deeply studied. This work is partly based on our previous experience with the Andorra-I CHIP extension. However, the current implementation is written in the concurrent logic programming language KL1, instead of C. The work challenges the assumption that committed choice logic programming languages cannot deal with don't know non-determinism. We have designed a process-based dynamic forking model to realize the non-deterministic choice in CLP(FD). Our first prototype, the CHUKL system, has been implemented in KL1 running under KLIC. We have been able to obtain some initial results to confirm the feasibility of the model.

1 Introduction

In recent years, Constraint Logic Programming (CLP) has become an increasingly important research area. CLP languages extend logic programming with efficient methods for solving constraints on specific domains, thus providing a practical way to solve many kinds of problems, especially those involving combinatorial search. Many CLP languages and systems have been developed; perhaps the most successful in practice have been those based on finite domains, in which variables can be constrained to range over a finite set of possible values. The first finite-domain CLP language was CHIP [12], but this has been followed by others, including clp(FD) [4] and cc(FD) [11], among others. Here we shall use the name CLP(FD) to refer to a generic finite-domain CLP language.

Some work has been done on parallel implementation of CLP(FD) languages, including [5] and [14], but these efforts have mostly exploited or-parallelism. The issues involved in or-parallel CLP languages are similar to those in the or-parallel implementation of Prolog. It is arguably more important to exploit and-parallelism in CLP languages: i.e., to solve all conjoined constraints in parallel, a form of "dependent and-parallelism". This is the type of parallelism exploited in committed choice concurrent logic programming languages, such as KL1. However, most problems that are solved well by CLP(FD) languages require search, which is not directly implemented in committed choice languages. Therefore, it has been suggested [1, 7] that and-parallel CLP(FD) languages could be implemented by extending a language that incorporates both concurrency, as in KL1, and search, as in Prolog. Some such languages have been designed, e.g., Pandora [1] and AKL [8]; one, Andorra-I Prolog [10], has even been extended with CLP(FD) primitives [7, 17]. The latter is the first CLP(FD) system that exploits both and-parallelism and or-parallelism.

The work presented in the paper is to study the and-parallel and or-parallel implementation of CLP(FD) in the concurrent logic programming language KL1.

1.1 Motivations

Motivation 1: One motivation of our work is similar to that of the Andorra-I CHIP extension [17], that is, to exploit full parallelism in CLP(FD). However, we selected a different approach from Andorra-I: we use KL1, a logic programming language, rather than the language C. To answer why KL1 leads to two other motivations of our work.

Motivation 2: For many years, committed choice logic programming languages have been facing a major criticism from Prolog programmers, that is, they lack don't know non-determinism. One research direction is to design new concurrent logic programming languages which incorporate this don't know non-determinism. There have been several new languages proposed, such as Pandora [1] and AKL [8], which constitute a family of non-deterministic concurrent logic programming languages. Another direction is to convert a Prolog program to a committed choice logic program through a programming transformation [15, 9, 13, 16]. However, these have not been very successful in practice because they usually work only for a restricted subset of Prolog, and usually do not provide and-parallelism. Our interest is to investigate the possibility of transforming non-deterministic concurrent logic programming languages to committed choice concurrent logic programming languages. CLP(FD) can be viewed as a member of the new language family because it requires non-deterministic choice and concurrency (or coroutining). An important feature of CLP(FD) is that its non-deterministic choice may be restricted to a single case: that is, to make choices by using the different values in domain variables. Therefore, we took CLP(FD) as the first step to study.

Motivation 3: An efficient implementation of KL1 in C (KLIC) [2] now exists for Unix machines, and a parallel version will soon be developed. This will provide an implementation of CLP(FD) that can run on distributed parallel architectures.

1.2 Overview of the paper

The rest of the paper is organized as follows. First, we present the execution model which underlies our implementation. Next, we describe the system itself, named CHUKL. We then discuss some initial results. Finally, a short conclusion and a list of future work ends the paper.

2 The Process-based Dynamic Forking Model

The computation of CLP(FD) comprises two alternating phases:

- the constraint propagation, and
- non-deterministic guessing.

We should first make it clear that, when parallelizing CLP(FD), these phases should not be performed in parallel. This is because, if the guessing phase goes ahead before the constraint propagation is completed, the same computation will be carried out in different branches. Thus, in order to avoid this duplicated computation, the two phases must be sequenced. This does not restrict the parallelism. The real parallelism exists within each phase:

- In the first phase, we can allow all constraints to work in parallel to achieve and-parallelism.
- In the second phase, several or-branches can be forked corresponding to the different possible value in the domain.

Although our previous work on the Andorra-I CHIP extension [7, 17] has already established a lot of experience in how to parallelize CLP(FD), Andorra-I's execution model can support CLP(FD) very easily. In Andorra-I, there are two execution phases: the deterministic and non-deterministic phases, which perfectly match CLP(FD)'s computation. Moreover, the choicepoint making and goal suspension mechanisms are already provided in Andorra-I.

To realize the above two forms of parallelism in KL1, there were several new implementation issues. The two main problems that need to be overcome first are discussed in the next two sections.

2.1 How to deal with non-deterministic forking

The first problem is how to achieve non-deterministic forking in a committed choice framework. As we discussed above, the constraint propagation phase can be naturally fitted into KL1 very well: the constraints can be realized as KL1 processes and executed in parallel. The non-deterministic phase is the problem. First of all, we need to detect when all constraints suspend, i.e., reach their fixed point. The next question is how to achieve the forking.

Our approach to solving this problem is originally inspired by a scheme of implementing Pandora under Parlog proposed by Gregory [6]. In Gregory's scheme, a special metalevel call is required:

```
metacall(Goal, FinalState)
```

The metacall executes the given `Goal`, then returns the `FinalState` of the execution, which is either "succeeded", "failed" or "suspended". In case of "suspended", a list of suspended goals is attached. By using this metacall in a top level interpreter, we can achieve the Andorra execution model on committed choice languages. When the deterministic execution terminates, a list of suspended goals is returned to the interpreter. One of the suspended goals may be selected to make a choicepoint, the rest of them can then be copied for forking.

The above scheme gives a high-level union (merge) between committed choice languages and don't know non-determinism. However, the scheme has not been realized because the metacall is rather too complicated to implement efficiently in practice. In KL1, a metacall, called "shoen", is provided which can report the final state of a given goal as "succeeded", "failed" or "suspended", but the suspended goals are not returned. Moreover, in the case where the computation is suspended (i.e., deadlocked), the "shoen" only detects it through garbage collection. In JAM [3], a shared memory implementation of Parlog, the deadlock is detected instantly, but the suspended goals is obtained by a exhaustive search of the heap, which is a very expensive operation. It is clear that, in order to provide a metacall as required in [6] efficiently, we have to substantially increase the complexity of the abstract machine and sacrifice performance.

Thus, the target of our approach is not to rely on any metacall but add our own control to achieve similar functionality. The idea is as follows. An extra process, called the controller, is introduced. All constraint processes are connected with the controller, which monitors the whole computation. Each constraint works independently until it reaches the fixed point. It then sends a "suspend" message to the controller and waits for a reply. The controller will then detect the situation.

If it is not deadlock, the controller sends a "retry" reply to the constraints; the constraint propagation phase then carries on again. If it is deadlock, this means that the constraint propagation phase has terminated. Then the next step, the non-deterministic choice, is handled by the controller in the following way.

First, the controller selects an unbound domain variable, reads out the possible values from the variable's domain and puts them into a list, `ValList`.

The next thing is to fork N or-branches corresponding to the N different values in `ValList`. To achieve this, the controller spawns N new controller processes corresponding to the N different branches, then it sends a messages "`fork(ValList,NetList)`" back to all constraints, where `NetList` is a list of communication channels corresponding to the newly spawned controllers. This is needed for the constraints to re-establish a new network with new controllers.

When a constraint process receives this reply, it then spawns N new processes corresponding to `ValList` and `NetList`. The guessed domain variable (if used in the constraint) is replaced by one of the values in `ValList`.

2.2 How to Represent Domain Variables

The most essential information attached to a domain variable is a set of the current possible values in the domain.

In existing (low level) Prolog based implementation, the domain variable is represented as a special data structure in which all necessary information are attached, such as what the current possible values are and which constraints are dependent on the domain variable. Within a constraint, each domain variable is actually a pointer to its domain data structure. When the constraint needs to update a domain, the operation is like a destructive assignment in procedural languages: we overwrite an already instantiated value by a new value. For example, when the constraint `X > 3` is executed while `X` ranges from 0,...,9, the old interval 0,...,9 is replaced by a new interval 4,...,9.

In low level implementations, we have freedom to directly implement this kind of destructive assignment. It can be done directly on each individual value without changing any other part of the domain data structure.

However, in logic programming languages, it is impossible to adopt the same method. The updating of a domain must be done by reconstructing the domain structure[1]. The problem is that a domain variable is normally shared by several constraints; when one constraint updates the domain by creating a new structure, we have to allow all constraints to access the most recent

[1]In the latest KLIC implementation, a new feature called "generic objects" is provided. This is an interface to allow a user to add new data objects to the KLIC system. All operations on the new data objects are defined by the user and written in C. We believe that this facility can be applied to the domain data structure, and we are planning to utilize it in the near feature when the interface and the extension are more stable.

values of the domain. Another issue relates to the and-parallel execution. When the constraint propagation phase is running in parallel, we have to guarantee that updating a domain is an atomic operation. This is achieved in the Andorra-I CHIP extension by locking a domain when accessing it. But, in KL1 and other logic programming languages, there is no primitive to allow us to lock a data structure.

Our solution is to use a process to represent a domain variable instead of using a data structure. Each domain variable is a KL1 process. It manipulates the domain and stores all necessary auxiliary information. In each constraint process, for each domain variable there is a communication channel to the corresponding domain variable process. When a constraint needs to update a domain, it sends a message through the communication channel to the domain variable process. The actual updating is then done by the domain variable process, not the constraint process. Therefore, atomic operation is guaranteed.

2.3 The Summary of the Model

To characterize our computation model, we call it the *process-based dynamic forking model.* It is process-based because every object in the system is represented as a process. We have three kinds of processes:

- the constraint process (named **constraint solver**),
- the domain variable process (named **domain manager**), and
- the control process (named **controller**).

We use the word dynamic forking because the non-deterministic forking is carried out at run time dynamically. Note that the or-parallel copying of the environment is done in a distributed way: each process is in charge of generating its own copies.

3 The CHUKL system

CHUKL is a first prototype implementation based on the above model. The general picture of the system is shown in Figure 1.

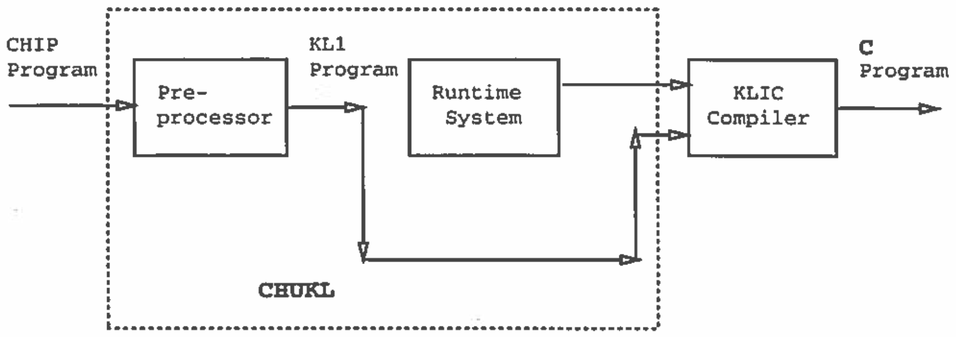


Figure 1: CHUKL: THE SYSTEM CONSTRUCTION

CHUKL has two components: a preprocessor and a runtime system.

The preprocessor takes a CHIP program as input, and translates it into a KL1 program which can be run by linking it with the runtime system. All basic primitives to solve the finite domain constraints are supported by the runtime system. By applying the KLIC compiler, we can finally obtain C code.

At the moment, to concentrate on trying out the main idea, we impose some restrictions on the input CHIP program which make the implementation much easier. First, we assume that choicepoints in the given CHIP program must only be made by labelling (i.e., the **indomain** builtin). Second, we impose a restriction that all domain variables' declarations must be made at the beginning of a program. Finally, we only support some basic arithmetic and symbolic constraints; that is, the more advanced builtin primitives like "minmax", "if-then-else" and "heuristic labelling" are not implemented.

3.1 The Preprocessor

The transformation from CHIP program to KL1 is mainly a source-to-source translation. For example, the following defines a list of domain variables in CHIP:

```
ListVar :: Min..Max
```

This is translated into a KL1 goal:

```
init_domain(ListVarGround, Min, Max, Network)
```

where `ListVarGround` is the ground representation of `ListVar` and the extra argument `Network` is needed to deal with the communication between constraints and controller.

The following is a simple example. We list both the CHIP program and the translated KL1 program.

```
% CHIP version

main:-
    [X,Y] :: 0..9,        % (1) domain declaration
    X + Y #= 4,           % (2) setting up constraints
    X #> 2,               %
    Y #< 3,               %
    indomain(X),          % (3) labelling
    indomain(Y).          %

% KL1 version

main:-
    init_domain([x,y], 0,9, DvarNet),            % [X,Y] :: 0..9
    add_controller_net(DvarNet, Cnet, Net0),
    create_arith_const(c1, x+y=4, Net0, Net1),   % X + Y #= 4
    create_arith_const(c2, x>2,   Net1, Net2),   % X #> 2
    create_arith_const(c3, y<3,   Net2, Net3),   % Y #< 3
    remove_controller_net(Net3, FinalDvarnet),
    create_controller(Cnet, FinalDvarnet).       % indomain(X), indomain(Y)

PROGRAM 1: a very simple example
```

Another job done by the preprocessor is to normalize linear equality and inequality. A linear equality/inequality is normalized if the following two conditions hold:

- all variables only appear once; and
- no minus sign appears in an equality or inequality.

For example, `x+y-3 = 2*x` is not a normalized equality. Its normalized form is `y = x+3`.

Obviously, both the translation and normalization explained above are rather trivial work. In particular, the input CHIP programs are currently restricted. Thus, translating CHIP to KL1 is very straightforward. At moment, we only implemented the normalization part, but omitted the translation part. We believe that if we remove some of the current restrictions, for example, to allow a choicepoint to be created by the CHIP program, or to allow the user to declare a domain variable anywhere, the preprocessing will become more difficult. These issues will be investigated carefully as the next step.

3.2 The Runtime System

The runtime system realizes the execution model described in last section.

Three kinds of processes, **controller, domain manager** and **constraint solver** are invoked by calling the goals `create_controller`, `init_domain` and `create_const` which are generated by the processor (as shown in Program 1).

One issue we have not discussed is how to wake up a suspended constraint after its domain variable's value is updated. Normally, this is done in the following way. For each domain variable, we store all constraints which are dependent on the variable. Some are dependent on the domain's minimum and/or maximum value; others are dependent on whether the domain has a unique value left. When the domain changes, we can then follow the information in the domain variable to resume the corresponding constraints. This is the most effective scheme, because the constraints can be reactivated immediately when necessary. However, in the current CHUKL system, we have not applied the scheme. The main concern is to avoid a complicated stream network between the domain variable processes and the constraint processes. The essential communication between constraints and domain variables is to read or update a domain. This requires a two-way communication between the constraints (clients) and the domain variables (server). If we let a domain variable process deal with the constraints' suspension and resumption, a separate communication network is needed. It will increase the complexity of the system substantially. As the network between processes has to be relinked dynamically after each forking, it is rather important to make the network as simple as possible. Therefore, we designed the following simple scheme. In each domain variable, we keep a list of constraints that have read the current domain value. When all constraints suspend, the controller checks whether all domain variables' current values have been read by all dependent constraints; if so, the propagation phase is terminated, otherwise, the control process tells the constraints to retry. In this way, some and-parallelism might be lost, but it simplifies substantially the network between the processes. As we particularly want to try out the main idea, dynamic non-deterministic forking, the simplicity of the system is more important.

Next we use the simple example (Program 1) given in the last subsection to illustrate the execution of CHUKL's runtime system.

In Figure 2, there are 6 diagrams to show the major execution stages of the given example. In order to distinguish the times at which the various operations take place, we use **t1, t2, t3** and so on to indicate the sequence of the operations. In the controller, there are two counters: one for suspended constraints and another for unsolved constraints. They are indicated as "**const**" and "**sus**" in Figure 2.

As marked in Figure 2, there are three sets of two-way communication streams connected between the processes and working for different purposes:

- **stream 1** is mainly used for the constraint solver to send a request to a domain manager to read or update the domain values. Back communication is needed for both cases. In the case of read, the current domain value is returned. In the case of update, if the updated value is consistent with the current domain, the domain manager replies "true" to the constraint, otherwise, the reply is "false".

- **stream 2** is between a constraint solver and the controller. When a constraint solver is either created, suspended, solved or failed, it reports its state to the controller through this stream. In case of suspension, back communication from the controller is required. This reply is either "retry" or "fork".

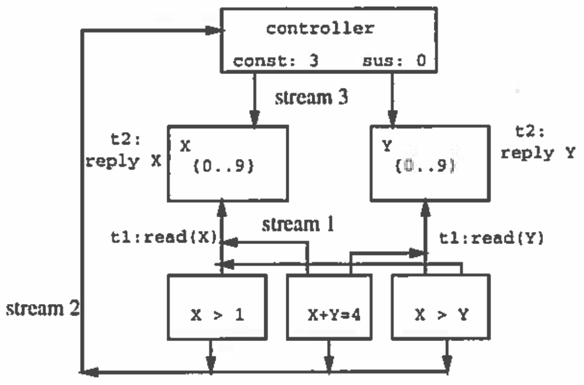
- **stream 3** is between the controller and a domain manager. In the non-deterministic choice phase, the controller needs to read a selected domain variable's values in order to make a fork. Moreover, in our current implementation, the controller needs to check whether all domains have been read by the constraints in order to detect deadlock.

Figure 2(a) shows the initial state of the execution. The first operation is that each constraint solver sends a **read** message to the domain managers, then the initial value of the domains X and Y are returned. Thus, each constraint now keeps a non-shared local copy of domain variable X and/or Y.

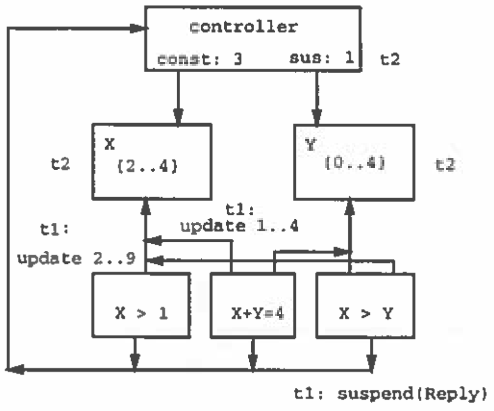
In the next step, which is shown in Figure 2(b), three constraints are first executed independently in parallel. X > Y suspends without any change to X and Y. A message **suspend(Reply)** is then sent to the controller. The counter of suspended goals is increased from 0 to 1. Meanwhile, the other two constraints are propagating effectively through a partial lookahead algorithm. They generate reduced domain values for X and Y, and send the new values back to the domain manager. As there is no conflict between the new and old values, the two domain managers modify their values, and reply "true" to the constraint solver indicating that the update is valid. Note that the value 2..4 is a result of merging two updates 2..9 and 0..4.

The constraint propagation is still carrying on in Figure 2(c). X > 1 is now solved after successfully changing domain X's minimum to 2. It sends a **solved** message to the controller. The counter of the total number of constraints is decreased from 3 to 2. X+Y=4 carries on its partial lookahead computation, and generates an even smaller domain for Y. Finally, it sends **suspend(Reply)** to the controller. Now the number of unsolved constraints and the number of suspended constraints are equal, which indicates that the constraint propagation phase can be terminated.

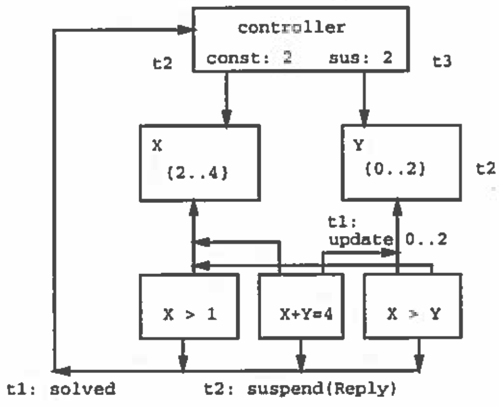
Figure 2(d) shows how the CHUKL system handles deadlock. When the controller modifies its two counters it checks whether they reach the same value. If so, it sends a message **check_all_readby** to all domain variables. In our example, the constraint X > Y has not read the latest updated values, so the controller instructs it to retry using the new X and Y. X > Y



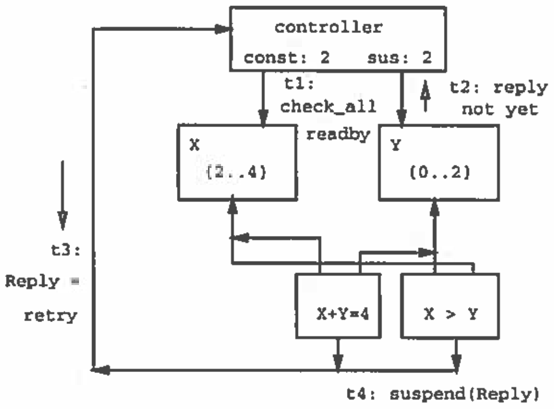
(a). read domains



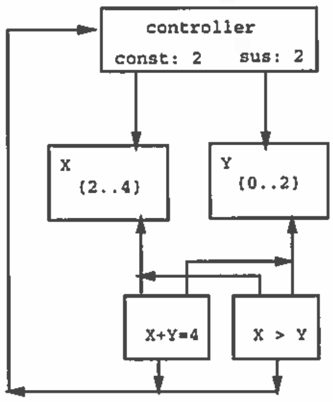
(b). update domains



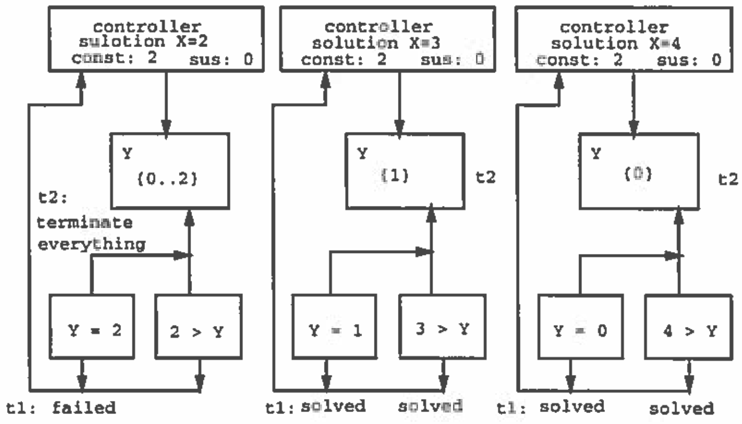
(c). keep updating till the fixed point



(d). handle the deallock



(e). fork or-branches



(f). find solutions

Figure 2: CHUKL: AN EXAMPLE

was suspended on variable `Reply`; as long as `Reply` is instantiated to `retry`, it becomes active again. Unfortunately, the constraint cannot propagate any further, so it suspends again.

In the next step, shown in Figure 2(e), the controller notices that its two counters are again equal; it then does the same checking as step (d) (this operation is not displayed here to save space). This time, all domain managers report that they have been read by their dependent constraints, so the deadlock is confirmed. To achieve forking, the controller sends a message to the `X` domain manager, to read out all possible values in `X`: 2, 3 and 4. It then forks itself to three new controllers; meanwhile it sends a fork message to the constraint solvers and the `Y` domain manager.

Figure 2(f) shows the status after forking. Three independent or-branches are created, in which `X` is replaced by 2, 3 and 4. All communication streams are dynamically established. These three branches can then be executed in parallel. The left branch fails immediately, but the other two branches produce solutions "`X = 3, Y = 1`", and "`X = 4, Y = 0`".

4 The Current Status and Some Initial Results

Our first prototype system, CHUKL, is under development. We have completed the most essential part of the runtime system in KLIC, and partially implemented the preprocessor, which is temporarily written in Prolog. We have been able to test several CHIP programs, among them one program which has 15 equations with 19 domain variables.

To verify the model's feasibility, we need to look at the following two aspects:

- can CHUKL achieve good performance?
- can the memory consumption be limited to a reasonable level?

As we are implementing the system in a high-level language rather than C, we expected that CHUKL's basic speed would be much slower than an efficient low-level implementation. For example, to update a domain's value only requires a few lines in C while, in CHUKL, we probably need about 30 reductions. We anticipated that the speed of CHUKL would be around several hundred times slower than the commercial CHIP system. Surprisingly, when we took a few typical benchmarks, a linear equation solving problem and two cryptarithmetic puzzles (send+more=money and roller-wheels=skoda), we found that CHUKL is only about 64 times slower on average (shown in Table 1)[2]. We have not been able to compare more benchmarks because the correct labelling order at the moment is not yet completely supported. In these three benchmarks, we added some simple control to make the labelling order exactly the same as CHIP's. We expect that, in the near future, we will be able to complete the labelling primitive, and then make more comprehensive comparisons.

We believe that the basic speed can be enhanced greatly through the following optimizations. First, there are many inefficiencies in our current implementation. For example, to simplify the problem, the controller currently always wakes up all constraints instead of just waking up the relevant ones. This is very wasteful and can easily be improved. Second, KLIC will soon provide a new feature: "generic objects". By using this facility, we can add new data structures and

[2]As we don't have the CHIP system, the CHIP results are tested using the latest version of CHIP at Essex University on a Solbourne workstation, which is half the speed of our Sparc.

program name	Time in milliseconds		H/W
	CHIP4 Solb	CHUKL SunSparc	adjusted ratio
send-more-money	10/2	120	24
roller-wheel-skoda	1810/2	67410	74
equations-15-19	199810/2	9360000	93
average			64

Table 1: THE BASIC SPEED IN CHIP4 AND CHUKL

define all operations on the structure in C. Domain variables can then be implemented more efficiently using this technique.

Another aspect to evaluate concerning our system is its memory consumption. As committed choice logic programs do no backtracking, one major doubt was whether the huge search space created in CHUKL is manageable. A positive result which we have found so far is that all of our benchmarks are able to run in a 4Mbyte heap, among these the largest program runs for over two hours. This shows that the garbage collection in KLIC is efficient, and the memory usage of our model is indeed manageable.

5 Conclusion and Future Work

In this work, we have proposed a process-based dynamic forking model to achieve non-deterministic choice in committed choice languages. A prototype system, CHUKL, has been implemented in KLIC. The initial results have shown that the basic performance of CHUKL is very promising. Compared with CHIP, which is a well refined low-level implementation in C, we are only about 64 times slower, which is quite an encouraging result as a first prototype implementation. Moreover, the memory usage is acceptable. These have confirmed the feasibility of the model.

As future work, there are three aspects, as follows.

First, we need to look at the speculative scheduling issue. At the moment, in CHUKL, no control is provided between or-branches yet. Therefore, we can only deal with all-solutions programs.

Second, we need to enhance the builtin primitives. In particular, the following three important language features need to be provided: a predicate to achieve branch-and-bound algorithms, "minmax"; the implication combinator, "if-then-else"; and predicates to support the heuristic search.

Finally, as mentioned above, we plan to use KLIC's "generic objects" to optimize the basic data structures and we will also optimize the system by removing the inefficiency in the current implementation.

Acknowledgements

The development of the CHUKL system is supported by ICOT as part of the FGCS Follow-on Project.

References

[1] R. Bahgat and S. Gregory. Pandora: Non-deterministic Parallel Logic Programming. In *Proceedings of the International Logic Programming Conference*, June 1989.

[2] T. Chikayama, T. Fujise, and H. Yashiro. A Portable and Reasonably Efficient Implementation of KL1. In *Proceedings of the International Logic Programming Conference*, June 1993.

[3] J. Crammond. Implementation of Committed Choice Logic Languages on Shared Memory Multiprocessors. Research Report PAR 88/4, Dept. of Computing, Imperial College, London, May 1988.

[4] D. Diaz and P. Codognet. A Minimal Extension of the WAM for clp(FD). In *Proceedings of the International Logic Programming Conference*, June 1993.

[5] M. Dorachevsky and A. Veron. Binding Techniques and Garbage Collection for Or-Parallel CLP System. Technical report ECRC-92-21, ECRC, August 1992.

[6] S. Gregory. Pandora: Parlog + Andorra. Presentation Given At *The Gigalips Project Workshop*, March 1988.

[7] S. Gregory and R. Yang. Parallel Constraint Solving in Andorra-I. In *International Conference on Fifth Generation Computer Systems*, June 1992.

[8] S. Janson and S. Haridi. Programming Paradigms of the Andorra Kernel Language. In *Proceedings of the International Logic Programming Symposium*, October 1991.

[9] A. Okumura and Y. Matsumoto. Parallel Programming with Layered Streams. In *Proceedings of the International Logic Programming Symposium*, October 1987.

[10] V. Santos Costa, D.H.D. Warren, and R. Yang. Andorra-I: a parallel Prolog system that transparently exploits both and- and or-parallelism. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, April 1991.

[11] V. Saraswat, P. Van Hentenryck, and Yves Deville. *Design, Implementation and Evaluation of the Constraint Language cc(FD)*. Technical Report No. CS-93-02, Dept. of Computer Science, Brown University, Jan 1993.

[12] H. Simonis, A. Aggoun, T. Graf, M. Dincbas, P. Van Hentenryck, and F. Berthier. The constraint logic programming language CHIP. In *International Conference on Fifth Generation Computer Systems 1988*, ICOT, 1988.

[13] H. Tamaki. Stream-based Compilation of Ground I/O Prolog into Committed Choice Non-Deterministic Languages. In *Proceedings of the International Logic Programming Conference*, June 1987.

[14] B.M. Tong and H.F. Leung. Concurrent constraint logic programming on massively parallel SIMD computers. In *Proceedings of the International Logic Programming Symposium*, Vancouver, October 1993.

[15] K. Ueda. Making Exhaustive Search Programs Deterministic. In *Proceedings of the International Logic Programming Conference*, June 1986.

[16] K. Ueda. Making Exhaustive Search Programs Deterministic, Part II. In *Proceedings of the International Logic Programming Conference*, June 1987.

[17] R. Yang, A. Beaumont, and D. Munaf. Programming Style Guide for Andorra-I CHIP Extension. Technical Report, Dept. of Computer Science, Bristol University, August 1993.

Compilation of Concurrent Declarative Languages

Z. M. Ariola, B. C. Massey, M. Sami, and E. Tick
Dept. of Computer Science
University of Oregon

March 3, 1994

Abstract

The plethora of concurrent declarative language families, each with subtly different semantics, makes the design and implementation of static analyses for these languages a demanding task. However, many of the languages share underlying structure, and if this structure can be exploited, static analysis techniques can be shared across language families. These techniques can thus provide a common kernel for the implementation of quality compilers for this entire language class.

The purpose of this paper is to exploit the similarities of non-strict functional and concurrent logic languages in the design of a common intermediate language (CIL). The CIL is introduced incrementally, giving at each step the rationale for its extension. As an application, we present, in CIL form, some state-of-the-art static partitioning algorithms from the literature. This allows us to "uncover" the relative advantages and disadvantages of the analyses, and determine promising directions for improving static partitioning.

1 Introduction

The compilation of concurrent declarative languages, such as Id and Strand, for high-performance multiprocessors depends fundamentally on increasing task granularity (partitioning) and hiding various latencies such as memory accesses. The plethora of language families, each with subtly different semantics, makes the design and implementation of static analyses, the backbone of a good compiler, a demanding task. However, many of the languages share underlying structure, and if this structure can be exploited, static analysis techniques can be shared across language families.

Multithreading at the processor level is required to correctly implement concurrent languages that support non-strict control constructs and synchronizing data structures. The higher thread granularities lead to lower process creation, management, and communication overheads. Multithreading also provides the ability to tolerate long and unpredictable communication and synchronization latencies, as the processor can switch to another ready thread rather than wait for a response. The process of identifying portions of a program that can be executed as threads, referred to as *partitioning*, has been a subject of growing interest to the implementors of concurrent logic programming languages and functional languages, here referred to collectively as concurrent declarative languages.

In this paper, the important issues concerning partitioning both concurrent logic programming languages and non-strict functional languages are presented, and some of the important partitioning techniques are discussed. The similarities between non-strict functional languages with single-assignment features and committed-choice languages with logic variables lead us to believe that partitioning techniques applicable in one language domain will be successful in the other. The partitioning algorithm and its associated static analyses are the kernel of any high-performance compiler for concurrent declarative languages. We believe that a good first step in understanding the state-of-the-art in partitioning and how to improve it is to define a common intermediate language (CIL) for both non-strict functional and concurrent logic languages, and evaluate partitioning algorithms with respect to the CIL.

This paper is organized as follows. Section 2 reviews the family of concurrent declarative languages. Section 3 defines a CIL covering a slightly restricted subset of this family. Section 4 reviews the partitioning problem and the two major types of algorithm proposed in the literature, using a CIL program as a comparative example. This comparison reveals strong and weak points in each approach, and suggests directions for future exploration. A summary of the research, conclusions, and discussion of future work is given in Section 5.

2 Review of Concurrent Declarative Language

This paper focuses on the compilation of concurrent declarative languages containing what we refer to as *dynamic-single-assignment (DSA) variables.* This family includes Id [20], the parallel version of Haskell [22, 5], Strand [10], and many others. DSA variables correspond to the notion of "I-structures" of Id [6], and to the "logic variables" of Strand. A DSA variable begins its life unbound, and may be bound only once during program execution. However, in contrast to static-single-assignment variables, the point at which a DSA variable is bound is not in general statically determinable: binding can occur at any time during program execution. Attempts to read a DSA variable before it is bound will typically cause the reader to suspend. Once bound, attempts to rebind a DSA variable will "fail," and will typically cause execution of the entire program to fail.[1]

In this section, we review two languages with DSA variables, chosen from the two language families of principle interest in this paper: the concurrent functional language Id, and the concurrent logic programming language Strand. We note some of the similarities between these languages, and thus motivate the introduction of our CIL.

Id is a higher-order, non-strict, strongly-typed functional language, augmented with I-structures [6] and M-structures [7]. I-structures are arrays whose elements get "refined" during the course of computation: that is, it is possible to create an I-structure without giving a definition for each of its elements. (In a purely functional language a variable is given exactly one binding or definition at creation time.) For our purposes, single-element I-structures are DSA variables [19]. M-structures add non-determinism to the languages; in this paper, however, we will not address this feature. Id has non-strict semantics, that is, a function is invoked even though its parameters may not be completed evaluated yet. However, unlike other modern functional languages, non-strictness is not implemented in terms of lazy evaluation. Id has a *parallel evaluation strategy*, that is, all computations are performed in parallel, not merely those whose results are sure to be needed. Thus, the parameters of a function are all evaluated in parallel with the evaluation of its body, even though the body might not need some parameter. For example, consider the following Id definitions:

$$\text{Def } nth\ (x : xs)\ n = \text{If } n == 1 \text{ then } x \text{ else } nth\ xs\ (n-1)$$
$$\text{Def } Ints\ x = x : (Ints\ (x+1))$$

and the main expression: $nth\ (Ints\ 1)\ 5$. (Note: The expression $x : xs$ denotes a list whose head is x and tail is xs.) The function nth is invoked even though the evaluation of the first parameter does

[1] With regard to logic programs, we restrict ourselves to committed-choice languages with "eventual" tell unification [24] or assignment, rather than "atomic" tell unification, thus implying that binding failure causes program failure.

not terminate. The evaluation of the main expression will return 5 as a result. This is accomplished by evaluating the expression *Ints* 1 in parallel with the body of the *nth* function. Note that the main expression returns a value even though its evaluation does not terminate. In this specific example, we can safely assume that 5 is the final answer, however, in the general case, due to the presence of single-assignment variables, the result must be interpreted as a partial answer, that is, the answer could be refined further. In particular, as mentioned before, the partial answer could be refined to failure in case of multiple bindings; a condition that cannot be checked statically.[2]

All variables in a Strand program are DSA variables, although some may be bound immediately upon creation. Syntactically, a program consists of a sequence of procedures, each of which consists of a sequence of *clauses.* Each clause is of the form "*Head* :- *Guard* | *Body*" where the head is a term and the guard and body are sequences of terms. Execution is defined by *goals* corresponding to procedure invocations, also called tasks and processes. Each goal specifies a procedure and its arguments: the head and body of each clause of the procedure are matched against the goal. If one or more clauses match the goal, one clause is selected for *commitment.* The committed clause's body will consist of a (possibly empty) sequence of new goals, which are all evaluated concurrently. If a goal fails to match because not enough information is available, the goal will suspend. If the goal fails to match for other reasons, the goal will cause failure of the program. A program may also fail by *deadlock* which occurs when all goals have suspended. Finally, as mentioned above, a program may fail if a DSA variable is assigned to a second time, thus informally there are three sources of program failure.

There are marked similarities among the members of this language family. Execution is described in terms of *eager concurrency*: all program units will be executed concurrently as soon as they are reached, the only source of delay being suspension due to an attempt to read an unbound DSA variable. The concurrency in the languages is very fine-grained — many concurrent tasks will be spawned, and each of these tasks will be relatively small, such as a goal or function evaluation. Finally, the fact that tasks suspend when trying to read an unbound DSA variable implies that the languages are subject to the possibility of deadlock, and thus the semantics are truly concurrent. If an attempt is made at compile time to decide the order of execution of tasks (*e.g.,* for uniprocessor execution), care must be taken to ensure that some latter task will not provide a binding which some earlier task needs to proceed.

This problem of static scheduling to increase task granularity while avoiding deadlock is a main component of compilers for executing these languages on high-performance multiprocessors [30, 15, 23, 11, 21, 18]. Comparative analysis of different partitioning techniques is best achieved by first developing an intermediate language common to the DSA language family. A common language allows us to accurately compare currently proposed analyses, as well as develop a single, unified analysis technique. The latter will help concurrent declarative language researchers to avoid unconsciously duplicating past research. The advantage of defining an intermediate *language* as opposed to an intermediate *form* (*i.e.,* a non-executable structure) is that the operational semantics of the language can assist us in formalizing and showing correctness of the static analyses.

The common intermediate language (CIL) or *kernel* language presented attempts to draw together the functional and logic programming communities. By sharing a CIL foundation, it will be easier to share ideas and implementations, exchange programs and reduce harmful diversity. However, we do not seek uniformity. There are clear differences between these languages. But having a common framework will allow us more easily to highlight and study these differences. In the next section, we develop the

[2]The reader may refer to Ariola and Arvind [2] for further clarification. The parallel evaluation of Id has been adopted by the newly developed pH language, a parallel variant of the Haskell language [13]. Haskell is increasingly the standard language in the non-strict functional language research community.

```
sum(Xs, S) ← sum'(Xs, 0, S).                                  §1

sum'([ ], S, S).                                              §2
sum'([X | Xs], P, S) ← plus(X, P, P'), sum'(Xs, P', S).       §3

plus(0, 0, 0).
plus(0, 1, 1).
plus(1, 0, 1).
plus(2, 0, 2).
plus(2, 1, 3).
  ⋮
```

Figure 1: Sample Prolog Program

```
sum(Xs) → sum'(Xs, 0)                                 §1
sum'([ ], P) → P                                      §2
sum'([X | Xs], P) → { P'  =  plus(X, P)               §3
                      S   =  sum'(Xs, P')
                       in  S }

plus(0, 0) → 0
plus(0, 1) → 1
plus(1, 0) → 1
plus(2, 0) → 2
plus(2, 1) → 3
  ⋮
```

Figure 2: Sample Program in Kernel Language

CIL incrementally, with a rationale for each extension.

3 Common Intermediate Language

In this section we introduce the main features of our kernel language through examples. In particular, we are interested in capturing the essential features needed to support DSA variables. At the outset we assume that a CIL program produces a unique result, independently of evaluation order. Thus, for instance, we disallow Prolog programs that produce multiple solutions, and concurrent logic procedures with non-mutually exclusive guards that are truly indeterminate, *e.g.*, stream mergers (which can easily be relegated to system-defined builtins).

Both concurrent logic languages and functional languages have gone through some evolution (and deevolution [27]) to support both parallelism and determinacy. Functional languages, on the one hand, restrict the handling of state, while concurrent logic languages restrict unification and backtracking. Consider the Prolog program in Figure 1.

Given a list of numbers, sum computes the sum of its elements. Two selection rules are in play: clauses are selected top-down, and body goals are selected in left-to-right order. Backtracking undoes bindings that lead to local and nonlocal failures. In this example, even if the body goals of sum′ were reversed, or the clauses of plus were reordered, a query with a ground input list would be guaranteed by backtracking to produce a correct solution (although possibly very inefficiently!)

We can rewrite the above example program in a committed-choice language, removing backtracking and restricting the clause selection to matching. In Parlog [8] or Strand the program remains textually identical to the one in Prolog, with the inclusion of *mode declarations*: sum(?,?), sum′(?,?,^), and plus(?,?,^). The input mode '?' indicates that the passed parameter must be bound and the '^' indicates that the argument binding is produced by the procedure itself. Other committed-choice languages indicate these *ask* and *tell* unifications in syntactically different ways, but the idea is the same.

In functional language terminology, we say that plus is *strict* in both of its input arguments. We can translate the above clauses into our kernel language as shown in Figure 2.

A typical query is written as $\{Res = \mathsf{sum}([1,2]) \text{ in } Res\}$. The kernel language semantics allows a variable to appear on the left-hand side of only one equation (the output instance), and on the right-hand side of multiple equations (the input instances). The key point enabling this translation is the determination of variable modes. This is difficult to do in many cases, and impossible in general for an unrestricted language, as we discuss in the next section.

Note that the kernel language is essentially a *graph rewriting system* [1, 4]: each clause is translated into a rewrite rule. Note also that in order to respect the communication pattern expressed in a logic program via logical variables, the kernel language must allow assignment of a name to each computation. For example, in the atom $\mathsf{plus}(X, P, P')$, P' denotes the result of the operation. We make this explicit by saying $P' = \mathsf{plus}(X, P)$. Thus the right-hand side of our rewrite rules are not just simple terms, as in rule §1, but can be a set of equations of the form $\alpha_1 = t_1, \cdots, \alpha_n = t_n$, where each variable is distinct from each other. The in keyword denotes the result of the term: for example, the result of rule §3 is S. Note that in the term $\{\alpha_1 = t_1, \cdots, \alpha_n = t_n \text{ in } \alpha_i\}$ the order of the equations is irrelevant. We discuss graph rewriting semantics in Section 3.2, after first dealing with mode analysis.

3.1 Mode Analysis

In logic programs, including committed-choice programs, no syntactic distinction is made between variables and their subterms which have *output mode* and are thus being bound in an occurrence, and those which have *input mode* and are thus being read in an occurrence. Thus, some form of mode analysis (*e.g.,* [28, 32]) or mode declarations [26] are required for translation into the kernel language. However, these methods are not perfect. For example, array elements can be aliased by indexing accesses, which means we might not be fully knowledgeable about all modes in an array. Furthermore, without good general aliasing analysis, these unknown array modes could "contaminate" the remainder of the program (to date, no mode analyzers do the sort of aliasing analysis required to effectively control this contamination).

Another point is that a certain class of automatic mode analyzers require language restrictions based on the concept of *permissible modes* [32, 25]. Sundararajan [25] defines a static analysis technique for determining a minimal set of permissible modes for a Prolog program. This is complicated by the lack of restrictions on Prolog unification. As discussed earlier, committed-choice languages restrict unification to ask and tell, making mode analysis considerably less complex. To further simplify the analysis, Ueda and Morita [32] restrict the language family to a "fully-moded" subset wherein each logic variable can have only a single producer. Furthermore, the subset requires equal modes in corresponding argument positions of a procedure's defining clauses (and all corresponding subterms *within* those arguments). These restrictions enable efficient, practical mode analysis [28].

Note that in the previous example, while in the concurrent logic program the processes spawned in clause §3 communicate through the logical variable P', in our setting the communication is rendered via a data dependency. This provides a hint for later phases of compilation that it is safe to sequentialize plus and sum$'$. However, if the implementation desires to execute plus(X, P) and the call to sum$'$ in parallel then P' will have to be implemented as a DSA variable. The key point is that instead of restricting programs to use only DSA variables (as in concurrent languages), programs written in the kernel language may explicitly indicate points where synchronization is not needed to respect the language semantics. The kernel language implementation may still eliminate some synchronization points through static analysis, as well as inserting synchronization points if desired.

There are programs that cannot be consistently moded, and thus cannot be translated into the kernel language as it now stands. Furthermore, even fully-moded programs can have variable occurrences in which some portion of the occurrence's term structure is read, and another portion written. This condition also precludes translation. We discuss an extension to deal with this in Section 3.4, after first discussing the basic rewriting semantics.

3.2 Rewrite Semantics

Let us now present the execution of a query to the example of the previous section, following a rewrite semantics. We start out with the term $\{Res = \mathsf{sum}([1,2])\ \mathsf{in}\ Res\}$, by applying rule §1 we obtain:

$$\{Res = \mathsf{sum}'([1,2],0)\ \mathsf{in}\ Res\}$$

by applying rule §3 we have:

$$\begin{array}{llllll} \{ & Res & = & \{ & P'' & = \mathsf{plus}(1,0) \\ & & & & S' & = \mathsf{sum}'([2],P'') \\ & & & & & \quad \mathsf{in}\ S'\} \\ & \mathsf{in}\ Res\} & & & & \end{array}$$

By pattern matching sum$'([1,2],0)$ with the left-hand side of rule §3 we have created the bindings $X = 1$, $Xs = [2]$: we then create an instance of the right-hand side of the rule. This corresponds to the renaming done before matching in concurrent logic languages, and can also be seen as the allocation of a frame in the context of a procedure call. In the kernel language we want to eliminate all syntactic sugar, and thus we will consider the above program equivalent to:

$$\begin{array}{llll} \{ & Res & = & S' \\ & P'' & = & \mathsf{plus}(1,0) \\ & S' & = & \mathsf{sum}'([2],P'') \\ & \mathsf{in}\ Res\} & & \end{array}$$

That is, the internal nesting of blocks does not matter. In the above program we can now execute, for example, the third expression, obtaining:

$$\begin{array}{llll} \{ & Res & = & S' \\ & P'' & = & \mathsf{plus}(1,0) \\ & S' & = & \mathsf{sum}'([\,],P''') \\ & P''' & = & \mathsf{plus}(2,P'') \\ & \mathsf{in}\ Res\} & & \end{array}$$

$SE \in$ Simple Expression

Variable	::=	$x \mid y \mid z \mid \cdots \mid a \mid b \mid \cdots \mid f \mid \cdots \mid x_1 \mid \cdots$
Constant	::=	*Integer* | *Boolean* | $\cdots$
SE	::=	*Variable* | *Constant*
Term	::=	$SE \mid \mathsf{f}^n(SE_1, \cdots, SE_n)$
Block	::=	$\{[Variable = Term]^* \text{ in } SE\}$
Definitions	::=	$Term \rightarrow Block \mid Term \rightarrow Term$
Program	::=	*Block*

Figure 3: The Grammar of Kernel_0

No conflict of names will arise, because at each step of the execution new names are introduced. Note also that $\mathsf{plus}(2, P'')$ cannot be executed until P'' gets a value, that is, until $\mathsf{plus}(1, 0)$ has executed.

There is a major difference between the computational model presented here and the one usually employed to give the operational semantics of concurrent languages. In our approach we do not keep a separate component describing the current value of each variable. Instead, these "substitutions" are kept directly in the term itself. Our preliminary kernel language is basically a system of recursive equations over a first-order signature Σ, as illustrated in Figure 3. Note that each functional symbol is applied to simple expressions only: that is, we assume that each expression has been given a name. It is possible to show that the term-rewriting system of Figure 3 is *confluent*, though this is outside the scope of this paper.

3.3 Multiple Values

Concurrent logic procedures can also return multiple values, as shown in the following example:

$$\mathsf{distribute}([\mathsf{send}(1, X) \mid Xs], Out1, Out2) \leftarrow Out1 = [X \mid Out1'], \mathsf{distribute}(Xs, Out1', Out2).$$
$$\mathsf{distribute}([\mathsf{send}(2, X) \mid Xs], Out1, Out2) \leftarrow Out2 = [X \mid Out2'], \mathsf{distribute}(Xs, Out1, Out2').$$
$$\mathsf{distribute}([\,], Out1, Out2) \leftarrow Out1 = [\,], Out2 = [\,].$$

In order to translate the above clauses we need to extend Kernel_0 by allowing multiple return values. The need for multiple-return-value facilities in intermediate languages has been illustrated in the context of functional languages [3]. A new language, Kernel_1, is obtained by changing the production of the syntactic category *Block* as follows:

$$Block ::= \{[Variable \cdots Variable = Term]^* \text{ in } SE \cdots SE\}$$

Thus the translation of the above program will be:

$$\begin{aligned}
\mathsf{distribute}([\mathsf{send}(1, X) \mid Xs]) \leftarrow \{\ & Out1 && = && [X \mid Out1'] \\
& Out1'\ Out2 && = && \mathsf{distribute}(Xs) \\
& && \text{in } Out1\ Out2\} \\
\mathsf{distribute}([\mathsf{send}(2, X) \mid Xs]) \rightarrow \{\ & Out2 && = && [X \mid Out2'] \\
& Out1\ Out2 && = && \mathsf{distribute}(Xs) \\
& && \text{in } Out1\ Out2\} \\
\mathsf{distribute}([\,]) \rightarrow [\,]\ [\,].
\end{aligned}$$

If a concurrent program is fully moded and furthermore, if no occurrence of a variable corresponds to multiple modes, then the program can be translated into a Kernel$_1$ program. In the next section, we tackle the problem of translating programs that do not meet these constraints.

3.4 Dynamic-Single-Assignment (DSA) Variables

Processes synchronize via DSA variables. However, they cannot make non-deterministic choices based on the availability of data. Consider the following program:

$$\begin{aligned}
&\mathsf{counter}(X) \leftarrow \mathsf{counter'}(X, 0).\\[1em]
&\mathsf{counter'}([\mathsf{clear} \mid Xs], C) \leftarrow \mathsf{counter'}(Xs, 0).\\
&\mathsf{counter'}([\mathsf{add} \mid Xs], C) \leftarrow C' := C + 1, \mathsf{counter'}(Xs, C').\\
&\mathsf{counter'}([\mathsf{read}(X) \mid Xs], C) \leftarrow X = C, \mathsf{counter'}(Xs, C).\\
&\mathsf{counter'}([\,], C).
\end{aligned}$$

Analysis can detect that the (first argument) stream is *input* and also the variable C is *input*; however, the stream contains further variables that need to be instantiated. Kernel$_1$ does not yet allow this, since we do not have the concept of an unbound variable. Instead, a variable is always associated with a particular value, and we do not have the possibility of allocating some variable and later initializing it. Thus, we now extend the kernel language with the concept of DSA variables. A DSA variable defines an unbound variable, it is allocated via the Mk-Var primitive. Associated with a DSA variable there are two operations: *read* and *write*. The *read* operation fetches the value of a DSA variable: however, if no value has yet been given to the DSA variable, the *read* is suspended until a *write* operation gives the variable a value. Moreover, if an attempt is made to *write* a DSA variable that has already been written, an error will be raised and execution of the program will fail. We thus extend the set of predefined function symbols with two new operators: Store of arity two (a DSA variable and the value to be stored), and Select, of arity one (the DSA variable). We may write $A = v$ instead of Store(A, v). Analogously, we may write A instead of Select(A). Finally, we will relax the restriction that a block must return a value, since some blocks may be invoked only for their side-effects. We use the special identifier "_" as indication that the expression does not return a meaningful result.

The translated program will be:

$$\begin{aligned}
&\mathsf{counter}(X) \rightarrow \mathsf{counter'}(X, 0).\\[1em]
&\mathsf{counter'}([\mathsf{clear} \mid Xs], C) \rightarrow \mathsf{counter'}(Xs, 0).\\
&\mathsf{counter'}([\mathsf{add} \mid Xs], C) \rightarrow \{\; C' \;=\; C + 1\\
&\qquad\qquad\qquad\qquad\qquad\quad \_ \;=\; \mathsf{counter'}(Xs, C')\}\\
&\mathsf{counter'}([\mathsf{read}(X) \mid Xs], C) \rightarrow \{\; X \;=\; C\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \_ \;=\; \mathsf{counter'}(Xs, C)\}\\
&\mathsf{counter'}([\,], C) \rightarrow \{\,\}.
\end{aligned}$$

For example, we can invoke the counter with the following expression:

$$\begin{aligned}
\{\; a &= \mathsf{Mk\text{-}Var}\\
b &= \mathsf{Mk\text{-}Var}\\
\_ &= \mathsf{counter}(\mathsf{add} : \mathsf{add} : \mathsf{read}\ a : \mathsf{add} : \mathsf{read}\ b : \mathit{nil})\}
\end{aligned}$$

As with Kernel$_1$, it is possible to show that the extended term-rewriting system is *confluent*.

Variable	::=	$x \mid y \mid z \mid \cdots \mid a \mid b \mid \cdots \mid f \mid \cdots \mid x_1 \mid \cdots$
Constant	::=	*Integer* | *Boolean* | $\cdots$
SE	::=	*Variable* | *Constant*
Term	::=	$SE \mid \mathrm{f}^n((SE)_1, \cdots, (SE)_n) \mid \mathrm{Select}(SE) \mid \mathrm{Mk\text{-}Var}$
Block	::=	$\{[Statement]^* \ \mathrm{in}\ SE \cdots SE\}$
Statement	::=	*Binding* | *Command*
Binding	::=	$Variable \cdots Variable = Term$
Commands	::=	Store(*SE*, *SE*)
Definitions	::=	$Term \rightarrow Block\ \ Predicate \mid Term \rightarrow Term\ \ Predicate$
Predicate	::=	$<(SE, SE) \mid >(SE, SE) \mid \cdots$
Program	::=	*Block*

Figure 4: The Grammar of Kernel$_2$

In order to express the notion of guards present in concurrent logic languages we need to further extend the kernel language with the concept of conditional rewrite rules. Thus for example, we will allow a rule to be expressed as: $l \rightarrow r \ \ P(x)$, where $P(x)$ is a "flat" predicate, *i.e.*, made of builtin predicates such as $\leq$. We will say that the above rule is applicable only if $P(x)$ is true. Note that all the free variables in $P(x)$ should be bound at the time $P(x)$ is evaluated, that is, no synchronization is assumed. The syntax of our final kernel language, Kernel$_2$, is given in Figure 4.

4 Partitioning Techniques

Compile-time partitioning of concurrent declarative languages is difficult because it must be both powerful enough to increase the thread size while avoiding the accidental introduction of deadlocking behavior. Problems arise because sometimes the correct order of execution of two processes can only be determined dynamically. Such processes have to be identified and placed in separate threads. This is especially intricate for concurrent logic programming languages, because of the prevalence of logic variables, which can cause hidden cycles through aliasing. Non-strict functional languages have analogous problems with hidden cycles through I-structures, which also require some scheduling decisions to be made at run time.

Several researchers have explored the issue of thread partitioning of non-strict functional languages [30, 15, 23, 11, 21]. While most of the techniques produce threads within a function and in isolation from the rest of the programs, the analysis of Traub *et al.* [31], referred to in this paper as *scanning analysis*, attempts to improve the thread size within a function by propagating dependence information from outside. Partitioning techniques have also been proposed in the context of granularity analysis for concurrent logic programming languages [9, 17, 29], most significant of which is the analysis proposed by King and Soper [16], referred to in this paper as *coloring analysis.*

The similarities between non-strict functional languages with I-structures and committed-choice languages with logic variables lead us to believe that partitioning techniques applicable in one language domain will be successful in the other. However, the techniques developed for different language families differ not only in their methodology but also in their input. For instance, scanning analysis requires the programs to be fully moded, and does not take into consideration the influence of aliasing. In contrast, coloring analysis does not require the programs to be fully moded and explicitly handles aliasing. The translation of both non-strict functional languages and committed-choice languages into the common intermediate language ($Kernel_2$, introduced in Section 3) provides a common framework for examining different analysis techniques. This facilitates "uncovering" the relative advantages and disadvantages of the current analyses, and determining future directions for developing an improved partitioning analysis.

The proposed techniques for partitioning also differ in the formalizations of the notion of "thread." This difference has significant influence on the results produced and the complexity of the techniques. It is thus important that the notion of "thread" used in each technique be clearly defined before a comparison can be made among them.

- *Liberal Thread:* A thread is defined to be a totally ordered set of atomic actions of a program. Every function may have one or more threads. Thus a thread can itself include calls to other threads, and also include *split-phase*[3] memory accesses, which implies that a thread may have to suspend and wait for other thread(s) to execute before it can continue.

  This definition is used commonly in partitioning concurrent logic programming languages, because it corresponds well to their operational semantics.

- *Conservative Thread:* A thread is defined to be a subset of the atomic actions comprising a program, such that: 1) the thread is contained within a single function; 2) a compile-time ordering can be determined for the thread which is valid in all contexts in which the containing function can be invoked, and 3) once the first instruction in a thread is executed, it is always possible to execute each of the remaining instructions, in the compile-time ordering without pause, interruption, or execution of instructions from other threads.

  This definition is used in partitioning "dataflow" languages. Synchronization only occurs at the start of the thread, and a thread, once scheduled, runs to completion. Thus, function calls and split-phase memory accesses result in multiple threads. Also, conditionals result in separate threads for each arm: they are regarded as a generalization of function calls.[4]

In order to compare and contrast the different techniques, a single example will be used as an illustration throughout the following sections. Figure 6 presents a contrived example expressed in the CIL.

4.1 Scanning Analysis

The *scanning analysis* technique proposed by Traub *et al.* [31] uses dependence and demand information to reduce modified forms of dataflow graphs. Their partitioning technique is based on a greedy algorithm which attempts to make the threads as large as possible. Traub *et al.* felt that this approach probably avoids the danger of over-sequentializing because of the limits on the thread size imposed by the conservative definition of the thread, the language model, the use of split-phase accesses, and the control paradigm. The program to be partitioned is expressed as a structured dataflow graph, which consists of a collection of acyclic directed graphs describing the *basic blocks* (roughly corresponding to a group of operators with the same control dependence) and *interfaces* which describe how the blocks relate to one another. Conditionals are treated as a generalization of function call. In this case, the interface relates a single caller (the enclosing expression) to multiple callees (the arms of the conditional). The predicate used to select an arm of the branch is encoded in the interface.

[3] Split-phase memory accesses involve separate request and response messages, may have long latencies, and on multiple requests, the responses may arrive out of order.

[4] There is in fact a third definition of thread, due to Grafe [11], which is conservative except that threads may contain branches. Synchronization occurs only at the start of the thread, and a thread, once scheduled, runs to completion. However, some instructions may be discarded depending on branch evaluations.

```
$f(A, Y_1, Y_2) \leftarrow \mathit{true}$ |
        $B_1 = X_3 + 1,$
        $B_2 = X_1 + 2,$
        $g(A, B_1, X_1, X_2),$
        $h(A, B_2, X_3),$
        $Y_1 = X_1 + X_2,$
        $Y_2 = X_2 + X_3.$

$g(A, B, X_1, X_2) \leftarrow A > 0$ |
        $A_1 = A - B,$
        $B_1 = A + B,$
        $g(A_1, B_1, X_4, X_5),$
        $X_1 = X_4 + 1,$
        $X_2 = X_4 + X_5.$

$g(A, B, X_1, X_2) \leftarrow A \leq$ |
        $X_1 = A - 2,$
        $X_2 = A + 3.$

$h(A, B, X_3) \leftarrow A > 0$ |
        $X_3 = A - 4.$

$h(A, B, X_3) \leftarrow A \leq 0$ |
        $X_3 = A - B.$
```

Figure 5: A Sample Strand Program

```
$f(A) \leftarrow \{$ $B_1$      $=$  $\mathrm{Select}(X_3) + 1,$
          $B_2$      $=$  $\mathrm{Select}(X_1) + 2,$
          $X_1\ X_2$ $=$  $g(A, B_1),$
          $X_3$      $=$  $h(A, B_2),$
          $Y_1$      $=$  $X_1 + X_2,$
          $Y_2$      $=$  $X_2 + X_3.$
                          $\mathit{in}\ Y_1\ Y_2\}$
$g(A, B) \leftarrow \{$ $A_1$      $=$  $A - B,$
             $B_1$      $=$  $A + B,$
             $X_4\ X_5$ $=$  $g(A_1, B_1),$
             $X_1$      $=$  $X_4 + 1,$
             $X_2$      $=$  $X_4 + X_5.$
                             $\mathit{in}\ X_1\ X_2\}\ A > 0.$
$g(A, \_) \leftarrow \{$ $X_1$  $=$  $A - 2,$
            $X_2$  $=$  $A + 3.$
                       $\mathit{in}\ X_1\ X_2\}\ A \leq 0.$
$h(A) \leftarrow \{$ $X_3$  $=$  $A - 4,$
                    $\mathit{in}\ X_3\}\ A > 0.$
$h(A, B) \leftarrow \{$ $X_3$  $=$  $A - B,$
                       $\mathit{in}\ X_3\}\ A \leq 0.$
```

Figure 6: The Sample CIL Program

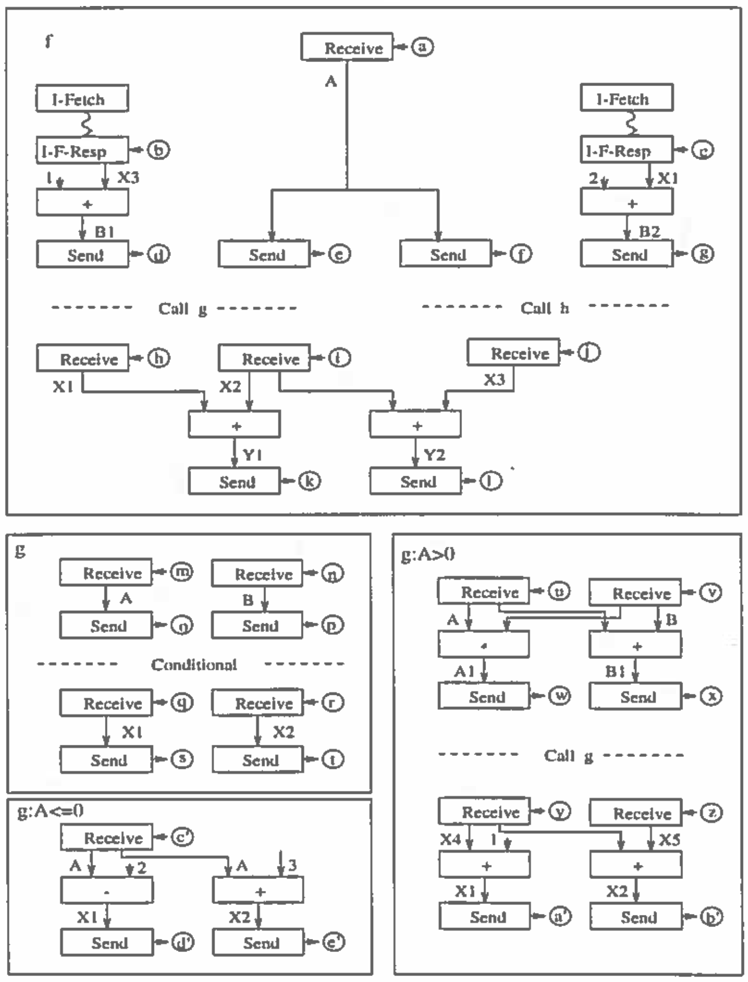


Figure 7: Structured Dataflow Graph Representation of the Sample Program

A CIL program can translated into this representation using a methodology similar to the one described by Nikhil [20]. Figure 7 shows the acyclic directed graph representation of the sample CIL program shown in Figure 6. The representation of h function is not given in Figure 7 due to space limitations. The vertices of the acyclic graphs describing the basic blocks represent primitive operators, and edges indicate dependences. These are augmented with annotations which encode information about some operators' behavior and interrelationships: these annotations are used extensively during analysis. In addition to the arithmetic and logical operators, the operator set includes *send* and *receive* operators which communicate values between basic blocks corresponding to function linkage and conditional expressions in the CIL. The *I-Fetch* and *I-Store* are synchronizing split-phase memory operators, where the response to an *I-Fetch* is not received until an *I-Store* (or a send) to the same location takes place. The response of an *I-Fetch* is received by an *I-F-Resp* instruction, which stands for *I-Fetch Response*. Every Select term in the source CIL program is replaced by an *I-Fetch* and *I-F-Resp* pair. Example of this can be seen in Figure 7. Similarly, a Store command in the source CIL program is translated as an *I-Store* operator.

Three forms of dependences are expressed in the graph representation: *certain*, *indirect*, and *potential*. A *certain dependence* is indicated by a straight edge between two nodes. An *indirect dependence* is a dependence completed through one or more vertices in a different basic block, and is represented by a squiggly edge. A *potential dependence* is an ordering constraint due to a data dependence that may differ depending on the context in which a function is invoked (say because of query), or which is otherwise unknown at compile time. All vertices that may be end-points of incoming potential dependences are annotated with an *inlet port* and those that may be outgoing potential dependences are annotated with an *outlet port*. These annotations are basically sets of names, linking potential corresponding outlets and inlets with each other.

The overall partitioning technique consists of the iterative application of two stages: *basic-block partitioning* and *global propagation*. We now explain each of these in turn.

4.1.1 Basic-Block Partitioning

Basic-block partitioning is used to group the vertices of a basic block into disjoint subsets that are trivially mapped into threads. The basic-block partitioning algorithm consists of *dependence-set* partitioning and *demand-set* partitioning. Dependence-set partitioning forms partitions by grouping together all nodes that depend on the same set of inlets. Demand-set partitioning groups together all nodes which are demanded by the same set of outlets. Since no distinction is made between certain and indirect edges, there may be two vertices in the same partition which are connected by an indirect edge. *Subpartitioning* is applied to divide such a partition into subsets so that the nodes connected by an indirect edge are in different partitions. Subpartitioning is accomplished during the same pass made by the dependence-set and demand-set partitioning algorithms.

The basic-block partitioning algorithm iteratively applies of dependence-set and demand-set partitioning with subpartitioning. This is done by considering each thread formed during a partitioning pass as a node of a reduced graph, and applying the next iteration to the reduced graph. Thus dependence-set and demand-set partitioning algorithms are repeatedly applied, forming a reduced graph at each stage, until the number of partitions does not change. A compile-time ordering of the operations can be obtained by any topological sort according to the dependences wholly within the thread.

We call this technique *scanning analysis* because the basic-block partitioning consists of a bidirectional scan of a function's body.

As an example of basic-block partitioning, consider the graph representation of function f in Figure 7. The dependence set of the addition operator which adds X_1 and X_2 to produce Y_1 is $\{h, i\}$. Hence

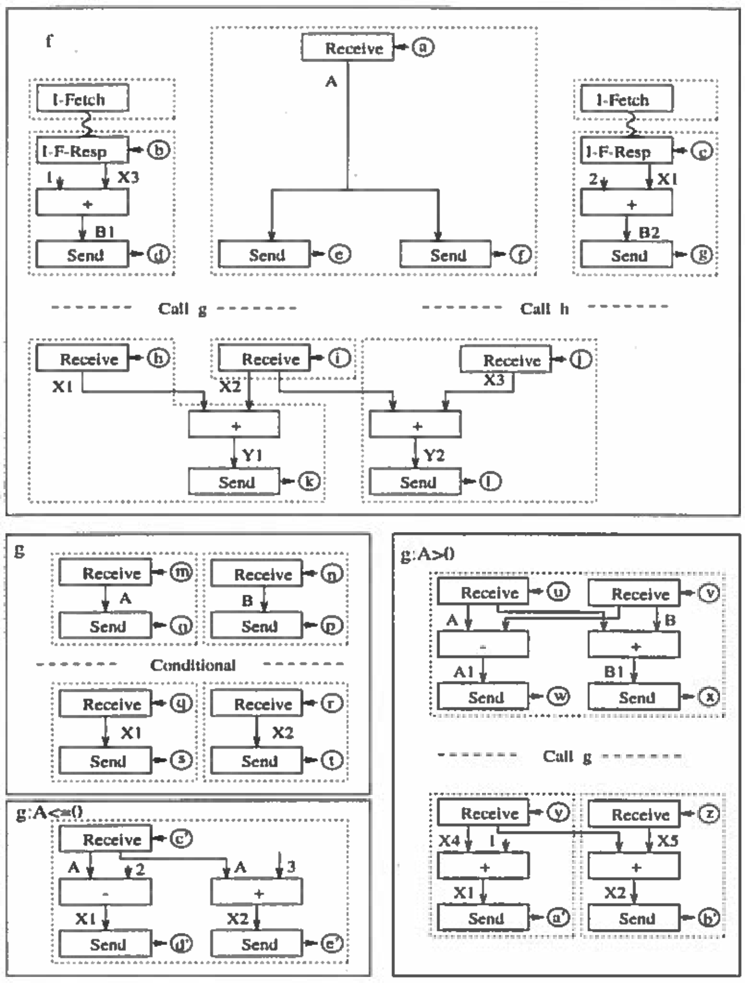


Figure 8: Partitions for the Sample Program

these two operators are placed in the same partition. In the next iteration, when demand-set partitioning is performed, the demand set of the *receive* operator which receives X_1, and that of the group of two operators formed earlier, is the same *i.e.*, $\{k\}$. This leads to all these operators being placed in the same partition, shown by a dotted box in Figure 8. Simple demand-set partitioning would place the *I-Fetch* operator which fetches X_3, in the same partition as the operators dependent upon it. Since a split-phase access cannot be placed in a conservative thread, subpartitioning is used to identify the presence of indirect edges in partitions and then separate such partitions across the indirect edge into two partitions.

4.1.2 Global Propagation

The effectiveness of the basic-block partitioning algorithm is limited by its inability to propagate dependence and demand information across control boundaries and functions. Global propagation derives dependence information across function boundaries and uses it to improve partitioning. The information gained during the partitioning of a function p may be used to form larger threads when partitioning another function q which calls p, in contrast to partitioning q in isolation. If q is known to be the only caller of p, then the partitioning information from q can likewise improve partitioning of p. This process is iterated to arrive at mutually improved version of both q and p.

As discussed by Traub *et al.* [31], scanning analysis, consisting of basic-block partitioning and global propagation, must be combined with a selection strategy to decide whether to repartition the basic blocks, or to globally propagate. One selection strategy is to form a call tree and alternately sweep from the leaves to the root and from the root to the leaves. At each node, representing a function composed of basic blocks, the node is repartitioned and the results are propagated alternately upwards or downwards.

If the program is recursive, a call *graph* rather than a call tree is required. However, the above selection strategy fails for a call graph because the existence of cycles makes it possible that the analysis will not terminate, i.e., a fixpoint may not be reached. In this case some spanning tree of the cyclic call graph may be chosen, and propagation limited to the interfaces in the spanning tree.

To illustrate, consider function g in Figure 8. No information can be propagated across the recursive call to g in the $A > 0$ arm. This prevents improvement of the partitioning of basic blocks across the recursive call because the dependence/demand information is propagated through the certain and indirect edges of the callee, which is function g itself.

4.2 Coloring Analysis

King and Soper's *coloring analysis* [16] is an example of a partitioning technique that uses information derived from abstract interpretation. In coloring analysis, two sets of constraints are used to generate threads. First, a thread must not contradict any data dependence in the program (this constraint is shared by scanning analysis). Second, terms of a program which are sufficiently coarse-grained should be allocated to different threads, because a thread may be evaluated in parallel with other threads. Coloring analysis was motivated by the serial combinator proposed as the grain for parallelism in the context of lazy graph reduction [12], which uses both data dependence information (obtained by strictness analysis) and granularity information to build threads.

The technique can be summarized as follows. The individual blocks of a program are considered in isolation. Alias analysis is used to identify shared variables. Since the modes of the variables are made explicit in the CIL, the analysis phase originally used [16] to infer the directionality of dataflow between variables is not required. A data dependence exists between two terms of a CIL function if one (or more) of the results produced by one term may be consumed by the other term. This idea is formalized by the data dependence relation Dep_c, which is a relation defined over $body_c$, the set of terms of a function c. A tuple of terms $< t_1, t_2 > \in Dep_c$ if t_2 depends on t_1. King and Soper [16] needed to define the data dependence relation $Dep_{c,p}$, for a given reduction sequence p. By abstract interpretation, they collected a conservative estimate of $Dep_{c,p}$ for each possible p, from which the data dependence relation Dep_c was constructed.

To illustrate these concepts, Figure 9 shows the functions f and g of the sample program with their terms labeled. Figure 10 presents the relation Dep_f as a cyclic directed graph.

A separation relation is derived from the data dependence to isolate pairs of terms which have to be allocated to different threads, because their data dependence can only be resolved at run-time. Pairs of terms which have to be separated in this way are indicated by a relation Sep_c on $body_c$ derived from Dep_c. Specifically, $< a_1, a_2 > \in Sep_c$ if $< a_1, a_2 > \in Dep_c$ and $< a_2, a_1 > \in Dep_c$. A similar separation relation $Gran_c$ on $body_c$ is also constructed based on the granularity of terms, so that two sufficiently coarse-grained terms can be placed in different threads. This is done by classifying terms as constant, linear, or non-linear depending upon the estimated cumulative difference between the growth of computation and

$$
\begin{array}{llcll}
\mathsf{f}(A) \leftarrow \{ & B_1 & = & X_3 + 1, & t(f,1) \\
 & B_2 & = & X_1 + 2, & t(f,2) \\
 & X_1\ X_2 & = & \mathsf{g}(A, B_1), & t(f,3) \\
 & X_3 & = & \mathsf{h}(A, B_2) & t(f,4) \\
 & Y_1 & = & X_1 + X_2, & t(f,5) \\
 & Y_2 & = & X_2 + X_3. & t(f,6) \\
 & & & \mathit{in}\ Y_1\ Y_2\} & \\
\mathsf{g}(A,B) \leftarrow \{ & A_1 & = & A - B, & t(g+,1) \\
 & B_1 & = & A + B, & t(g+,2) \\
 & X_4\ X_5 & = & \mathsf{g}(A_1, B_1), & t(g+,3) \\
 & X_1 & = & X_4 + 1, & t(g+,4) \\
 & X_2 & = & X_4 + X_5. & t(g+,5) \\
 & & & \mathit{in}\ X_1\ X_2\}\ A > 0. & \\
\mathsf{g}(A,\_) \leftarrow \{ & X_1 & = & A - 2, & t(g-,1) \\
 & X_2 & = & A + 3. & t(g-,2) \\
 & & & \mathit{in}\ X_1\ X_2\}\ A \leq 0. &
\end{array}
$$

Figure 9: Functions f and g of the Sample Program with Labeled Terms

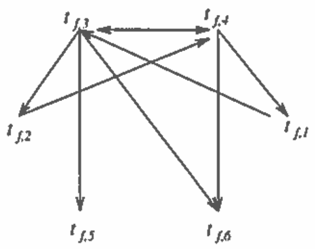
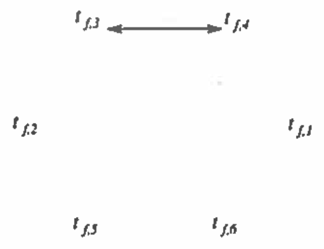
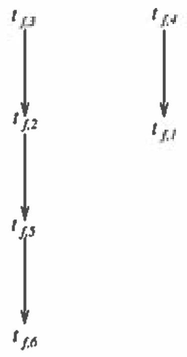


Figure 10: Data Dependence Relation Dep_f



(a) Sep_f



(b) Ord_f

Figure 11: Separation Relation and Total Ordering Relation for function f

communication during the execution lifespan. The constant class corresponds to predicates which are builtins, or are defined in terms of constant terms. The linear class corresponds to either non-recursive or linearly recursive predicates. These are fine-grained, whereas the non-linear class (neither constant nor recursive), corresponds to predicates which are typically coarse-grained. This classification gives a simple prescription for controlling granularity: two terms which are non-linear are allocated to different threads, whereas any other pair of terms can potentially be allocated to the same thread. Nonlinearities are indicated by a relation $Gran_c$ on $body_c$. $<a_i, a_j>\in Gran_c$ if a_i and a_j are nonlinear. For the example, $Gran_f$, $Gran_{g+}$, and $Gran_{g-}$ are empty due to the absence of nonlinear terms.

The aim of the analysis is minimization of the number of threads, and thus an increase in the proportion of terms which are ordered at compile time. Deriving an absolute minimum is NP-complete, but graph coloring can give good approximations for this problem. Partitioning $body_c$ corresponds to computing a k-coloring with respect to $Sep_c \cup Gran_c$, each color being interpreted as a separate thread. This leads to partitioning $body_c$ into $part_c = \{part_{c,1}, \cdots, part_{c,k}\}$, in which $a_1 \in part_{c,i}$ and $a_2 \in part_{c,j}$ for $i \neq j$ if either $<a_1, a_2>\in Sep_c$ or $<a_1, a_2>\in Gran_c$. To turn a partition $part_c$ into threads, a total ordering $o_{c,i}$ is assigned to each $part_{c,i}$, chosen not to contradict any data dependency in Dep_c. The cumulative effect of these total orderings is given by $Ord_c = o_{c,1} \cup \cdots \cup o_{c,k}$.

For function f of the sample program, application of the k-coloring algorithm with respect to $Sep_f \cup Gran_f$ results in $part_f = \{part_{f,1}, part_{f,2}\}$, where

$$\begin{aligned} part_{f,1} &= \{t(f,2), t(f,3), t(f,5), t(f,6)\} \\ part_{f,2} &= \{t(f,1), t(f,4)\} \end{aligned}$$

For function $g+$ of the sample program,

$$\begin{aligned} Dep_{g+} &= \{<t(g+,1), t(g+,3)>, <t(g+,2), t(g+,3)>, \\ &\quad <t(g+,3), t(g+,4)>, <t(g+,3), t(g+,5)>\} \\ Sep_{g+} &= \phi \\ Gran_{g+} &= \phi \end{aligned}$$

The application of the k-coloring algorithm with respect to $Sep_{g+} \cup Gran_{g+}$ results in one partition $part_{g+} = \{t(g+,1), t(g+,2), t(g+,3), t(g+,4), t(g+,5)\}$. A single partition $part_{g-} = \{t(g-,1), t(g-,2)\}$

is also obtained for the second clause of function g. These two partitions result in two threads after ordering: $o_{g+} = <t(g+,1), t(g+,2), t(g+,3), t(g+,4), t(g+,5)>$ and $o_{g-} = <t(g-,1), t(g-,2)>$.

4.3 Discussion

The two partitioning techniques explained in the earlier sections represent very different approaches developed for different programming language paradigms. To evaluate them it is necessary to understand the issues they address. These issues include number of the threads produced for a given program, prevention of deadlocks, simplicity, practicality, and the ability to tune the thread size to underlying hardware. As was mentioned earlier, the techniques differ not only in their methodology but also in their input. For instance, scanning analysis requires the programs to be fully moded, and does not take into consideration the influence of aliasing. In contrast, coloring analysis does not require the programs to be fully moded and explicitly handles aliasing.

The translation of programs from various language families into the CIL provides both techniques with the same starting point. This facilitates a fair comparison of the two techniques. It allows us to identify those phases of coloring analysisfor which there are no equivalent phases in scanning analysis. For instance, scanning analysisrequires the programs to be fully moded, and does not take into consideration the influence of aliasing. In contrast, coloring analysisdoes not require the programs to be fully moded and explicitly handles aliasing.

The number of threads for the sample program (Figure 6) obtained by coloring analysiswas 6, while for the same program scanning analysisproduced 20 threads. This difference in the number of threads is a result of many factors, most significant of which is the different definitions of "thread" used in the analysis. In the coloring analysis, a liberal thread can include more than one function, can have an arbitrary number of synchronizing and long latency accesses which may lead to suspensions, and can even contain references to other threads. In the scanning analysis, a conservative thread is a collection of instructions with a complete compile-time ordering. Procedure call boundaries, synchronizing and long latency accesses, and branches split threads. Synchronization occurs at the beginning of the thread, and once initialized the threads runs to completion without pause. For example, in the partitioning analysis, the requirement that split-phase accesses be kept in separate threads led to two more threads, and the calls to g and h in function f split threads as well. Another important reason for the difference in the number of threads is that, in coloring analysis, two disjoint nodes are kept in the same thread in arbitrary order, while scanning analysis will keep them in separate threads.

Thread definition has great influence on the analysis techniques chosen. Inclusion of more than one function in one thread has a potential of introducing deadlock if the scheduling order of the functions is determinable only dynamically. To prevent introduction of such deadlocking behavior, scanning analysis breaks threads at all function boundaries. Similarly, inclusion of synchronizing accesses in threads could potentially lead to deadlock if each of two threads block waiting for bindings which can only be provided by the other. In order to prevent any such possibility, scanning analysis splits threads whenever synchronizing access has to be done. Arms of conditionals are placed in separate threads, because there is a potential of cyclic dependences in this case as well.

Thus, the scanning analysis is based on a worst-case scenario. In real programs, not all functions require their scheduling order to be determined at run time, not every synchronizing access results in deadlock, and not every conditional produces cyclic dependences. Instead splitting threads only when not doing so would compromise the termination characteristics of the program, scanning analysis applies the same strategy under both safe and unsafe situations. This keeps the analysis technique simple on one hand, but results in small thread size on the other.

Moreover, there is no attempt to control the size of the thread. This is based on the realization that, given the strict definition of a conservative thread, there is practically no danger of over-sequentialization. In addition, the specialized hardware typically used for dataflow execution typically has low thread management overheads. However, to efficiently utilize a conventional multiprocessor, thread size may need to be controlled and load distribution may need to be performed.

In contrast, the coloring analysis uses explicit safety mechanisms to prevent potential deadlocks, by identifying terms which have to be kept in separate threads. In an effort to conservatively derive safe information, the abstraction techniques used lose information, potentially resulting in less granularity than might theoretically be exploitable. Serialization analysis employs a form of granularity analysis to control the size of the threads, which classifies terms as constant, linear, and non-linear. This scheme is very imprecise. As King and Soper [16] themselves mention, a large collection of techniques are used to analyze a small class of programs: thus, the difficulties of implementation in a compiler may outweigh the benefits achieved. For instance, all clauses of the example program are classified as either constant or linear, and as far as granularity constraint is considered, all terms of each function could be placed in a single thread.

The inclusion of multiple functions and synchronizing and long latency accesses in a thread does result in increased thread size. It can also result in inefficiency if overdone. Thread state has to be stored on every suspension and context switching has to be performed, which places a burden on the underlying hardware if suspension is frequent. Serialization analysis provides no control over the number of suspension points in a thread.

To summarize, scanning analysis is simple and practical, because it avoids safety checks by working under worst-case assumptions in all cases. The resulting thread size is small, even with the global nature of the analysis. In contrast, while coloring analysis includes safety checks to prevent unsafe partitioning, obtaining larger threads, some of the steps are complex and thus of questionable practicality. The analysis uses conservative and overestimated dependence information, potentially resulting in less granularity than can theoretically be exploited.

5 Conclusions and Future Work

Non-strict functional and concurrent logic languages seem to be strongly similar, although this has not been widely recognized by researchers in these communities. Thus, solutions to problems in one domain may well transfer to the other. In particular, the static analysis problems encountered during compilation of these languages seem to be quite general: comparable solutions to these problems have been adopted independently in the two language communities.

We thus suggest that translation of programs from both language families into a common intermediate language has a number of attractive features. In particular, we note that 1) A clear description of CIL semantics will permit easy definition of the semantics of traditional languages in terms of the CIL; 2) Optimizations performed on the CIL form will provide compilation improvements for both language families, and 3) Comparison between static analysis algorithms for CIL programs is much easier than comparisons between algorithms for static algorithms of programs written in a variety of different languages.

We propose a particular CIL, Kernel$_2$, as a candidate. We note that its semantics are easy to define via term rewriting, and that it seems to contain the important features necessary for the straightforward translation of non-strict functional and concurrent logic languages. While this language seems useful for illustrative purposes, one important area for future work is in obtaining a final CIL design which satisfies all of our criteria, yet is simple to understand, analyze, and implement.

By way of illustration of our technique, and because it is important in its own right, we discuss a particular static analysis example, that of partitioning or threading concurrent programs. A good partitioning strategy has to address several sometimes conflicting goals [14]. The thread length has to be maximized so as to decrease thread switches and explicit synchronization, and to increase pipeline utilization and locality. This however should not be at the expense of exploitable parallelism, as the objective of partitioning is to group program units when there is little or not easily exploitable parallelism. It is also necessary to identify portions of a program whose execution order is dynamically determined, so that they are not statically scheduled in a single thread. The thread size should be flexible: the compiler should be able to tune the size to the underlying hardware for efficiency.

We discuss two important thread-building algorithms, describing each by its action upon a sample Kernel$_2$ program. The comparison is made a great deal easier by the use of the CIL. We note some of the relative advantages and disadvantages of these techniques, and discuss their domains of applicability. The comparison emphasizes that different thread definitions, motivated by different needs, lead to different though comparable results of static analysis.

Future work in the development of partitioning algorithms will have to address the issue of obtaining optimal-sized threads based on a model that accounts for local *vs.* non-local execution and communication for a given architecture. The resulting threads can then be efficiently executed on architectures designed to execute threaded programs as well as on standard multiprocessors.

Acknowledgements

E. Tick was supported by an NSF Presidential Young Investigator award, with matching funds from Sequent Computer Systems Inc. M. Sami and B. Massey were supported by a grant from the Institute of New Generation Computer Technology (ICOT).

References

[1] Z. M. Ariola. *An Algebraic Approach to the Compilation and Operational Semantics of Functional Languages with I-Structures.* PhD thesis, Harvard University, June 1992.

[2] Z. M. Ariola and Arvind. P-TAC: A Parallel Intermediate Language. In *Conference on Functional Programming Languages and Computer Architecture*, pages 230–242. London, ACM Press, September 1989.

[3] Z. M. Ariola and Arvind. A Syntactic Approach to Program Transformation. In *ACM SIGPLAN Symposium on Parallel Evaluation and Semantics Based Program Manipulation*, New Haven, June 1991. Yale University, ACM Press.

[4] Z. M. Ariola and Arvind. Graph Rewriting Systems For Efficient Compilation. In *Term Graph Rewriting : Theory and Practice.* John Wiley and Sons, 1993.

[5] Arvind, L. Augusston, J. Hicks, R. S. Nikhil, S. Peyton-Jones, J. Stoy, and W. Williams. pH: a Parallel Haskell. Technical report, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, USA, September 1993.

[6] Arvind, R. S. Nikhil, and K. K. Pingali. I-Structures: Data Structures for Parallel Computing. In *Workshop on Graph Reduction*, number 279 in Lecture Notes in Computer Science, pages 336–369, Santa Fe, September/October 1986. Springer-Verlag.

[7] P. S. Barth, R. S. Nikhil, and Arvind. M-Structures: Extending a Parallel, Non-Strict, Functional Language with State. In *Conference on Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science, pages 538–568. Springer-Verlag, 1991.

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

[9] 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.

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

[11] J. E. Hoch, D. M. Davenport, V. G. Grafe, and K. M. Steele. Compile-Time Partitioning of a Non-Strict Language into Sequential Threads. In *Symposium on Parallel and Distributed Processing*, pages 180–189. Dallas, IEEE Computer Society Press, December 1991.

[12] P. Hudak and B. Goldberg. Serial Combinators: Optimal Grains for Parallelism. In *Conference on Functional Programming Languages and Computer Architecture*, pages 382–399, Nancy, France, September 1985. Springer-Verlag.

[13] P. Hudak, S. Peyton-Jones, and P. Wadler (Editors). Report on Programming Language Haskell: a Non-strict, Purely Functional Language, Version 1.2. *ACM SIGPLAN Notices*, 27(5), May 1992.

[14] R. A. Iannucci. Toward a Dataflow/von Neumann Hybrid Architecture. In *International Symposium on Computer Architecture*, pages 131–140. Honolulu, May 1988.

[15] R. A. Iannucci. *Parallel Machine Languages.* Kluwer Academic Publishers, Boston, 1990.

[16] 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.

[17] A. King and P. Soper. Schedule Analysis: A Full Theory, A Pilot Implementation, And A Preliminary Assessment. Technical Report CSTR 92–06, Department of Electronics and Computer Science, University Of Southampton, February 1992.

[18] 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.

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

[20] R. S. Nikhil. Id (Version 90.0) Reference Manual. Technical Report CSG Memo 284-a, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, USA, July 1990.

[21] R. S. Nikhil. A Multithreaded Implementation of Id Using P-RISC Graphs. In *Languages and Compilers for Parallel Computing*, 1993.

[22] R. S. Nikhil, Arvind, and J. Hicks. pH Language Proposal (Preliminary) — Draft. Technical report, MIT Laboratory for Computer Science, September 1993.

[23] K. E. Schauser, D. E. Culler, and T. von Eicken. Compiler-Controlled Multithreading for Lenient Parallel Languages. In *Conference on Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science, pages 50–72. Cambridge MA., Springer-Verlag, 1991.

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

[25] R. Sundararajan. *Data Flow and Control Flow Analysis of Logic Programs.* PhD thesis, Department of Computer Science, University of Oregon, 1994. In preparation.

[26] Z. Symogyi. A System of Precise Modes for Logic Programs. In *International Conference on Logic Programming*, pages 769–787. University of Melbourne, MIT Press, May 1987.

[27] E. Tick. The Deevolution of Concurrent Logic Programming Languages. *Journal of Logic Programming*, 1994. Invited for Tenth Anniversary Issue. In preparation.

[28] 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.

[29] 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.

[30] K. R. Traub. Compilation as Partitioning: A New Approach to Compiling Non-Strict Functional Languages. In *Conference on Functional Programming Languages and Computer Architecture*, pages 75–88, London, September 1989. ACM Press.

[31] 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.

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