PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1988,
edited by ICOT. © ICOT, 1988

COMPILE-TIME GRANULARITY ANALYSIS FOR PARALLEL LOGIC PROGRAMMING LANGUAGES

E. Tick*
Stanford University
Institute for New Generation Computer Technology

ABSTRACT

The paper describes a simple compiler analysis method for determining the "weight" of procedures in parallel logic programming languages. Using Flat Guarded Horn Clauses (FGHC) as an example, the analysis algorithm is described. Consideration of weights has been incorporated in the scheduler of a real-parallel FGHC emulator running on the Sequent Symmetry multiprocessor. Alternative demand-distribution methods are discussed, including *oldest-first* and *heaviest-first* distributions. Performance measurements, collected from a group of non-trivial benchmarks on eight processors, show that the new schemes do *not* perform significantly faster than conventional distribution methods. This result is attributed to a combination of factors overshadowing the benefits of the new method: high system overheads, the low cost of spawning a goal on a shared memory multiprocessor, and the increase in synchronization caused by the new methods. Directions of further research are discussed, indicating where further speedup can be attained.

1 Introduction

The problem of granularity in parallel architectures is the tradeoff between abundancy of parallel tasks vs. the overhead of managing the tasks. In parallel logic programming architectures based on conventional multiprocessor organizations (c.f., dataflow), it is best to create tasks of large granularity because this reduces communication, locking and management requirements, which are the weak points of conventional machines.

This paper describes and evaluates a method of static (compile-time) analysis to estimate the granularity of procedures in parallel logic programs. The idea is simple: a procedure's granularity is the sum of the granularities of the procedures its calls, and the granularity of a self-recursive clause (as in append) is one by definition. This estimate is called the procedure's *weight.* Compiled procedure calls of the program can then be annotated with the weights and a runtime execution mechanism can make scheduling decisions based on the weights.

To concretize the ideas, an implementation is done for the Flat Guarded Horn Clauses (FGHC, a dialect of which is called KL1) architecture called KL1-B [5]. FGHC is a stream-AND-parallel committed-choice language[11]. A static FGHC granularity-analyzer was written to estimate the granularity of each goal. These weights were attached to each procedure call. A real-parallel emulator for KL1-B, written by M. Sato [7, 8], was modified to store this weight with other procedure information during a procedure call. Various scheduling algorithms were then analyzed, each utilizing the weight information in an attempt to distribute only high-granularity goals. Although a dependent AND-parallel system is used to described these ideas, the concept of compile-time granularity analysis is general enough to be used for other parallel logic programming languages.

An FGHC clause is of the form:

$$H :- G_1, ... G_m | B_1, B_2, ..., B_n.$$

where H is the head of the clause, G_i are guards, "|" is the commit, and B_j are the body goals. Procedures are composed of sets of clauses with the same name and arity. Execution proceeds by attempting unification between a goal (the caller) and a clause head (the callee). If unification succeeds, execution of the guard goals are attempted. These goals can only be system-defined builtin procedures, e.g., arithmetic comparison. If the guard succeeds, the procedure call "commits" to that clause, i.e., any other possibly good candidate clauses are dismissed. If the head or guard fails, another candidate clause in the procedure is attempted (if all clauses fail, the program fails). In FGHC there is a third possibility also: that the call *suspends.* This is described in detail below.

FGHC restricts unification in the head and guard (the "passive part" of the clause) to be input unification only, i.e., bindings are not made. Output unification can be performed only in the body part (the "active part"). These restrictions allow AND-parallel execution of body goals and even OR-parallel execution of passive parts[1]

*Supported by NSF Grant No. IRI-8704576.

[1]The implementation discussed herein executes passive parts sequentially.

during a procedure call. Synchronization between processes is inherently performed by the requirement that no output bindings can be made in the passive part. If a binding is attempted, the call suspends. When the variable to which the binding was attempted is in fact bound (by another process), the suspended call is resumed. These semantics permit stream AND-parallel execution of the program, i.e., incomplete lists of data can be streamed from one parallel process to another in a producer/consumer relationship. For example, when a stream runs dry, the consumer receives the unbound tail of a list and suspends. When the producer generates more data, the consumer is resumed and continues processing the transmitted data. In the implementation discussed herein, these data structures all reside in shared memory.

The FGHC abstract execution model is a reduction mechanism wherein the initial user query (a set of goals) is reduced to the empty set. A single goal is reduced by unifying it successfully with a clause and then replacing the goal with the body goals of the matching clause. Reductions of goals can proceed in any order. Superimposed on this model is a suspension mechanism that causes goals to suspend and resume.

The real-parallel KL1 system[7] at ICOT consists of a compiler, assembler and instruction-set emulator for the Sequent Symmetry multiprocessor. This system is being used as a tool for designing the Parallel Inference Machine (PIM)[2]. One of the problems faced by this and other stream-AND-parallel architectures is *the overhead of exploiting small granularity parallelism.* Because all goals in the source language are parallel, no distinction is made in the architecture between goals.[2] In contrast is an OR-parallel system, such as Aurora [13], where goals are situated in an OR-tree. As Warren notes, heuristically, high-granularity goals are near the root of the OR-tree. In the AND-reduction model, however, no AND-tree exists to retain the information about granularity. Instead, for instance in the KL1-B system, a *goal pool* (implemented as a set of lists) holds the goals. We note however, that a goal list, which is pushed in a depth-first manner, is naturally ordered by the *age of goals* (this corresponds to the granularity information inherent in the OR-tree). We further note that given a weight for each goal, we can merge-sort each goal into the goal list in an attempt to gain a more accurate granularity order.

Hermenegildo[3] and Sato[8] discuss the advantages of an "on-demand" scheduling method wherein once a PE becomes idle, it either polls or sends a message to a busy PE, requesting a goal. The measurements presented by Sato were made by issuing the most recently pushed goal to the idle PE (we call this *youngest-first distribution*). This scheme has many advantages:

- **simplicity**— the goal list is implemented as a forward-linked chain of goal records. A procedure call (`enqueue_goal`) simply adds a new goal to the front of the list. Issuing a goal to an idle PE simply removes the current-most goal from the list (`dequeue_goal`).
- **low suspension**—pure depth-first scheduling effectively reduces the occurrence of suspensions because it utilizes the information inherent in the goal ordering given by the user.

The alternative of *oldest-first distribution* requires a backward-link for each goal record, and requires their runtime maintenance. In addition to this overhead, oldest-first distribution is not pure depth-first in the sense that although each PE is executing locally depth-first, all PEs are not cooperating to execute the program in a *combined* depth-first manner. Furthermore, *heaviest-first distribution* requires a weight value for each goal record, and requires their runtime comparison during procedure call, in order to merge-sort the record within the goal list. In addition to this overhead, heaviest-first distribution may cause undo suspensions because the user goal order may be subverted.

The pros and cons of these schemes are discussed in full detail within this paper. The first section of the paper describes how the compiler generates the goal weights. The second section describes how the goals are managed and distributed. In the final section performance measurements of these methods are presented and conclusions drawn.

2 Generating Weights

The first step in compile-time granularity analysis is to convert the program source into a cyclic call graph. For example consider the following minimal representation of a logic program, where letters represent goals.

```
:- perpetual m.

a.              m :- m.        c :- e,e,m,e,c.
a :- b,a,c.     m :- m.        c :- e,c.
a :- e,m,f.     m.             c.
a.              m.

e :- e.         b :- e,a.      f :- e.
e.              b.             f :- f.
```

The "perpetual" declaration is included specifically for procedures (in languages such as FGHC) which act as

[2]User-defined *pragma* are described and measured in Taki[9]. In this scheme, goals are distinguished by user annotations specifying the PE they should execute on. Such pragma are not easy to formulate and lessen the declarativity of the program. For the Queens benchmark, Taki gives two pragma annotated versions of 30 and 49 lines, based on a pure FGHC program of 14 lines. Furthermore, Taki's measurements were made on the Multi-Psi multiprocessor, which has no on-demand distribution mechanism, so the overheads of calculating pragma are not known.

objects, suspending until they receive an input message, and then waking-up, acting on the message, and suspending once again. These procedures are given zero weight. The simplified program and its graph are shown below.

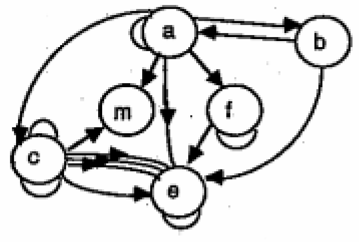
```
a :- b, a, c.
a :- e, m, f.

b :- e, a.

c :- e, e, m, e, c.
c :- e, c.

e :- e.

f :- e.
f :- f.
```



A node in the graph corresponds to a procedure. An arc represents a (potential) procedure call. Each arc is labeled corresponding to its clause. The unit of weight is one tail-recursive procedure, e.g., $w(\mathtt{e}) = 1$. The probability of executing any non-unit clause is considered equal. Thus $w(\mathtt{f}) = (1+1)/2 = 1$. The weight on any perpetual process (so declared because this cannot be determined syntactically) is 0, e.g, $w(\mathtt{m}) = 0$. Thus first clause of c has weight $1+1+0+1+1 = 4$, and the second clause has weight 2, therefore $w(\mathtt{c}) = (4+2)/2 = 3$. Calculation of $w(\mathtt{a})$ and $w(\mathtt{b})$ is difficult because they are mutually recursive. Here we chose one of the procedures to calculate first, marking it so. Assume this is a. When we need to recursively find the weight of b, as $w(\mathtt{b}) = w(\mathtt{e}) + w(\mathtt{a})$, we find that a has been marked. The weight of any marked procedure is defined to be 1. Thus $w(\mathtt{b}) = 1+1 = 2$, and $w(\mathtt{a}) = ((2+1+3)+(1+0+1))/2 = (6+2)/2 = 4$. Thus we assign the weights: $w(\mathtt{a}) = 4, w(\mathtt{b}) = 2, w(\mathtt{c}) = 3, w(\mathtt{e}) = 1, w(\mathtt{f}) = 1$.
The kernel of this algorithm, implemented in Prolog, is shown below.

```
% Node is node(ProcName, CallList, Marker, Weight)
%   CallList is [Clause1List,Clause2List,...]
%     Clause1List is [NodePtr1, NodePtr2,...]
%       NodePtr is a ptr to another Node
%   Marker is a flag used to cut recursion

p([]).
p([Node|Tail]) :- pProc(Node), p(Tail).

pProc(node(_,ClauseList,locked,Weight)) :-
    nonvar(Weight), !.
pProc(node(_,[],locked,0)) :- !.
pProc(node(_,ClauseList,locked,Weight)) :-
    pClause(ClauseList, Weight, 0, 0).

pClause([], Weight, N, PW) :-
    Weight is PW / N.
pClause([NodeList|Tail], Weight, N, PW) :-
    pNode(NodeList, 0, ClauseWeight),
    N1 is N+1,
    NewPW is PW + ClauseWeight,
    pClause(Tail, Weight, N1, NewPW).
```

```
pNode([], Weight, Weight).
pNode([node(_,_,Mark,W)|Tail], PW, FinalW) :-
    Mark==locked, !,
    (var(W) -> NewPW is PW + 1  % cut recursion
    ;          NewPW is PW + W),
    pNode(Tail, NewPW, FinalW).
pNode([node(P,L,Mark,W)|Tail], PW, FinalW) :-
    var(Mark), var(W), !,
    pProc(node(P,L,Mark,W)),    % child's weight
    NewPW is PW + W,
    pNode(Tail, NewPW, FinalW). % other siblings
```

This algorithm has the advantage of requiring only one-pass. However, because there are cycles in the call graph, the weights produces are somewhat inconsistent. For instance in the above example, $w(\mathtt{a}) = 4$ and $w(\mathtt{b}) = 2$. Yet b calls a! This difficulty arises because we cut a cycle in the graph at an arbitrary point (in this case, when initiating the calculation of $w(\mathtt{a})$ *before* $w(\mathtt{b})$). A possible improvement to this algorithm is a *relaxation phase* wherein a small number of passes, the weights are relaxed to make them more "consistent." Precise mathematical relaxation is not always possible, as illustrated most clearly by the following example[4]. In this program, there is no consistent assignment of weights to a and b, given $w(\mathtt{e}) \neq 0$.

```
a :- b, e.
b :- a, e.
e :- e.
```

In practice, minor inconsistencies in the weights do not affect performance. However, major inconsistencies, wherein a given goal is much heavier than surrounding goals that actually have larger granularity, can undermine goal distribution. The false heavy goal may be distributed ahead of an older, lighter goal. In this case, age is a much more accurate estimator of granularity.

The definition of weight as presented in the algorithm does *not* account for the number and type of procedure arguments, or their unification "strength," i.e., what work is required to commit the procedure. Instead, a heuristic is used wherein goals higher in the call hierarchy comprise more work than those lower in the hierarchy. For committed-choice languages this simple model of granularity appears to be as accurate as any other because stream communication has the effect of breaking up execution flow. For non-committed-choice languages without streams, such as independent AND-parallel Prolog, procedure input arguments are always complete at the time of procedure call. Therefore it may be possible to calculate (at compile time) the unification strength of the procedure invocation, and incorporate this into the granularity estimator [1]. Research is currently underway to develop such an abstract interpretation-based granularity analysis based on the domain of unification strength.

3 Goal Management

Important attributes of a good goal management policy are:

1. to reduce suspensions.
2. to make insertion (in the goal-list) of a body goal fast.
3. to favor spawning[3] of high-granularity body goals.
4. to make spawning of a goal fast.

To keep single PE execution fast, insertion of body goals in the goal-list must be efficient. To retain load balancing, high-granularity goals must be favored to be stolen by idle PEs. Finally, to keep spawning goals efficient, accessing goals by an idle PE must be fast. If the satisfaction of these constraints increases suspensions, performance gain may be lost. Note that Sato's scheme successfully meets points (1), (2) and (4), but not (3). Note also that as (3) improves, less emphasis is placed on (4) because as we retain better load balancing, less goals are spawned.

Pure depth-first scheduling with youngest-first goal distribution is illustrated below (a different example from the previous). The right-hand side represents a goal list, where the goals are represented by letters. The head of the goal list is leftmost. On the left-hand side, reduction transforms a goal into a body goal list, the goals of which (from right to left) are pushed onto the goal list.

```
          goal-list =    {a b c}
reduce a to {d e f} =>   {d e f b c}
reduce d to {}      =>   {e f b c}
reduce e to {a e}   =>   {a e f b c}
spawn a             =>   {e f b c}
reduce e to {}      =>   {f b c}
                    ....
```

In the KL1-B emulator, the `enqueue_goal` instruction adds a goal to the goal list. Note that in a depth-first evaluation, the goal is added to the top of the goal-list. This naturally sorts the goals by "age," which is similar to the heuristic in the OR-Parallel Prolog Aurora system of searching for a goal near the root of the OR-parallel tree. An *oldest-first distribution* scheme is illustrated below.

```
          goal-list =    {a b c}
reduce a to {d e f} =>   {d e f b c}
reduce d to {}      =>   {e f b c}
reduce e to {a e}   =>   {a e f b c}
spawn c             =>   {a e f b}
reduce a to {d e f} =>   {d e f e f b}
                    ....
```

[3]We use the term "spawn" to mean issuing a goal to another PE. We call local execution of goals, "reduction."

Now consider a scheme wherein each goal has a weight. We use a different abstract instruction, `enqueue_goal_with_priority`, to insert a weighted goal onto the goal list. We discuss here a few methods of insertion, trading time of insertion for accuracy of the sort, and trading age vs. weight. The discussion here represents only a sample of the many possibilities.

We first consider the simplest, and most expensive scheme: a perfect merge-sort. Here, the new goal weight is compared to each goal record in the goal list and always inserted in the correct position. This is illustrated below, where each goal is denoted by an integer representing its weight.

```
          goal-list  =   {0 0 1 2 8}
reduce 0 to {5 6 7}  =>  {0 1 2 5 6 7 8}
reduce 0 to {7' 8'}  =>  {1 2 5 6 7' 7 8' 8}
spawn 8              =>  {1 2 5 6 7' 7 8'}
                     ....
```

This method has the advantage of guaranteeing order, but suffers from high insertion overhead. We now discuss schemes that reduce this overhead.

Consider a scheme wherein the new goal is compared *only* to the last goal of the goal list. If the body goal is lighter, it is added to the front of the list. If the body goal is heavier, it is added to the end of the list. If the body goal is equal, it is inserted in the goal list before all other goals of equal weight, thus preserving age order in this instance. This method attempts to keep the heaviest goal at the end of the list. When the end goal is spawned on an idle PE, the new end-goal may not be heaviest; however, the end goal will soon be replenished with a heavy goal. Unfortunately this scheme, because it is only an approximate sort, often spawns younger rather than older goals. This scheme is illustrated below.

```
          goal-list  =   {0 0 1 2 8}
reduce 0 to {5 6 7}  =>  {5 6 7 0 1 2 8}
reduce 5 to {4 8'}   =>  {4 6 7 0 1 2 8' 8}
spawn 8              =>  {4 6 7 0 1 2 8'}
reduce 4 to {}       =>  {6 7 0 1 2 8'}
spawn 8'             =>  {6 7 0 1 2}
reduce 6 to {}       =>  {7 0 1 2}
spawn 2              =>  {7 0 1}
reduce 7 to {7 9}    =>  {7 0 1 9}
                     ....
```

We can extend this by limiting the number of comparisons permitted when attempting to insert the new goal. If the goal is still lighter than the *nth* goal from the end of the goal list, it is simply placed at the front of the list. For example, if $n = 3$:

```
          goal-list  =   {0 0 1 2 8}
reduce 0 to {5 6 7}  =>  {5 0 1 2 6 7 8}
reduce 5 to {7' 8'}  =>  {7' 0 1 2 6 7 8' 8}
spawn 8              =>  {7' 5 1 2 6 7 8'}
                     ....
```

In this case, the goals with weights 5 and 7' could not be merged in three comparisons, so they were added to the front of the goal list. We can optimize this method with a state variable pointing within the goal list, to the goal last compared. If the goals are pushed in order of decreasing weight, we can use this insertion pointer to avoid recomparisons. The disadvantage of this method is that additional overhead is needed to reset the insertion pointer between every reduction and goal spawning. The method is illustrated below with a limit of two comparisons. Note that now the goals of weights 5 and 7' can be correctly placed.

```
          goal-list  =   {0 0 1 2 8}
reduce 0 to {5 6 7}  =>  {0 1 2 5 6 7 8}
reduce 0 to {7' 8'}  =>  {1 2 5 6 7' 7 8' 8}
spawn 8              =>  {1 2 5 6 7' 7 8'}
                         ....
```

The scheduling methods discussed above stress efficiency of procedure call, as well as ensuring that the last goal of the goal queue always has large granularity. Three important issues have not been properly addressed.

- **suspension**— Such methods, because they deviate from depth-first scheduling, cause how much additional suspension, and does this overhead outweigh any performance gains? We note that in the programs analyzed to date, the user-defined goal order closely corresponds to the weight order, i.e., the goals are pushed in order of decreasing weight. If we guarantee this correspondence, by artificially zeroing body goal weights that disobey, then we can guarantee the goals are executed in a *depth-first-like* manner. This means that the goals are executed one after the other, but possibly with different goals interspersed, and possibly the last goals are spawned to another PE and executed out of order.
- **space**—Deviation from pure depth-first evaluation can cause large goal lists to build up. This impacts performance by reducing cache locality and decreasing garbage collection efficiency.
- **age**—Age-order is a better estimator of granularity than weight. Although we always guarantee placing equal weight goals in age precedence, we favor young, heavy goals over old, light goals. One problem is that age is not explicitly represented in these schemes, therefore a metric combining age and weight cannot be formulated.

4 Measurements

We have discussed youngest-first, oldest-first and heaviest-first goal distribution methods for KL1. These ideas were tested on a group of benchmark programs. Unfortunately, because the KL1 real-parallel system is under-development, only a few benchmarks could consistently run to completion. One requirement was to execute long-running, non-trivial programs that did not fall into any stereotypes. **Puzzle** is a solid-packing problem written in an object-oriented style. **Semigroup** calculates the members of a mathematical semigroup by using data streams and filters. **Waltz** is a line-drawing analysis problem using layered-streams [6]. **Triangle** is a board game playing program (generate & test) automatically translated from Prolog into FGHC using Ueda's continuation method [12]. For more details of these programs see Tick[10]. These programs all attain significant speedups on multiple PEs using the real-parallel KL1 system. Therefore measurements are presented here for eight PEs only.

Weights for these programs were calculated and attached to each procedure call in the assembly code, using `enqueue_goal_with_priority` instructions. The user defined goal order was *not* changed, except in the case of zero-weight goals which were always executed first. In almost all procedures, the user defined order corresponded to goals being pushed in order of decreasing weight, which is what is desired for these experiments. In the few instances where user-defined order contradicted weight order, we locally modified the weights to follow the user. The experiments measured are listed below.

1. depth-first scheduling, youngest-first distribution.
2. oldest-first distribution.
3. semi-sort: if new goal is heavier or equal to end goal, push in age order.
4. pure sort starting from last goal each enqueue.
5. limited sort ($n = 50$) starting from an insertion pointer each enqueue.

Table 1 gives the Sequent Symmetry execution times (in seconds, for eight PEs) for the various programs previously discussed. The statistics are averaged over the fastest three runs of each program. Garbage collections (GCs) varied: **Waltz** had no GCs, each **Semigroup** had 0–1 GCs and each **Puzzle** had 14–15 GCs. **Triangle** was eccentric because the *youngest-first* distribution (scheme 1) invoked 7–8 GCs per run, whereas the other distributions invoked only 3–4 GCs. Thus the new distribution schemes reduce garbage production by efficient scheduling. **Triangle**, partially because it was generated by automatic translation, has only one clause with more than one body goal. As a result, the weights are not useful because goals with single body goals are reduced immediately. Thus schemes 3–5 are equivalent to scheme 2.

	1	2	3	4	5
Puzzle					
E (sec)	105.83	101.33	102.07	104.13	102.53
$E_1 - E$	0.00	4.50	3.77	1.70	3.30
σ	0.17	0.93	0.37	0.19	0.90
E/E_1	1.00	0.96	0.96	0.98	0.97
Semigroup					
E	48.03	44.00	44.97	43.33	47.53
$E_1 - E$	0.00	4.03	3.07	4.70	0.50
σ	0.65	2.14	1.03	0.59	0.45
E/E_1	1.00	0.92	0.94	0.90	0.99
Waltz					
E	41.73	40.37	45.37	48.07	45.04
$E_1 - E$	0.00	1.37	-3.63	-6.34	-3.31
σ	1.77	0.82	0.50	0.98	2.53
E/E_1	1.00	0.97	1.09	1.15	1.08
Triangle					
E	430.93	416.00			
$E_1 - E$	0.00	14.93			
σ	1.57	1.30			
E/E_1	1.00	0.97			

Table 1: Performance of Distribution Schemes on 8 PEs (times in seconds)

The performance gains measured for the new distribution methods (Table 1, $E_1 - E$) fall outside of one standard deviation of the measurements. However, we do not observe any significant performance increase—at most 10% and in general 4% or less. **Puzzle** and **Triangle**, both all-solutions search problems, obtain only 3% speedup, partially because the default scheduler (scheme 1) is very efficient. For this type of program with a single, regular tree with large branching factor, the oldest-first method first distributes the goals near the root, giving high-performance. Schemes 3–5 erroneously push younger (high weight) goals onto the back of the goal-list for distribution, thus actually slowing down execution.

Semigroup, a determinate problem, achieves the best speedup possibly because it doesn't have a regular tree. **Waltz**, using layered-streams, shows some speedup for the oldest-first distribution (scheme 2), but other methods cause slowdown. This is because layered-streams programs are susceptible to suspensions and have almost no natural granularity, being a collection of many small filters.

We are confident that reduction of the implementation overheads and tuning of the new distribution schemes can save at least 2% of the total execution time. In addition, measurements on the current emulator, on all benchmarks, are skewed because of the large overhead of suspensions. Suspensions are slow because the architecture uses a single, general-purpose **suspend** instruction, instead of lower-level optimizations to avoid work during suspension. Also, the suspended variable is hooked to the *goal*, thus upon resumption, the entire procedure must be retried. A possible improvement is to hook the variable to both the goal and an entry point within the procedure. In any case, when suspension overhead decreases, the new distribution schemes presented here are expected to improve with respect to the original method. Since the new schemes are incurring a larger number of suspensions, a reduction in suspension/resumption overheads will benefit them most.

5 Conclusions

A method of estimating the granularity of procedure calls in parallel logic programming languages is described. This analysis method is statically done at compile-time in one pass and is therefore efficient. With the analysis are goal distribution methods such as *oldest-first* and *heaviest-first*, which attempt to distribute high-granularity goals to idle PEs in an on-demand basis. Preliminary performance measurements of these methods (for KL1) indicate that speedups of less than 10% are attained on eight PEs on a shared memory multiprocessor. There are several reasons for this. The Symmetry has a relatively low penalty for spawning a task so that minor improvements in scheduling do not cause major improvements in performance. Some of the all-solutions search programs measured have regular trees that can be efficiently scheduled by any method. The other programs measured have little natural granularity, using very fine-grained pipeline and layered-stream parallelism. In addition, the programs display critical timings wherein minor disruption of depth-first scheduling can result in a significant increase of suspensions.

We expect further speedups when the KL1-B suspension mechanism is improved, and the distribution methods are more efficiently encoded. In addition, we expect that larger applications programs (where large natural granularity is buried) will be able to better exploit these new distribution schemes because the weight metric more closely estimates granularity than the age metric in large programs. The programs measured here are still small enough that they have fairly regular trees, wherein age is a better estimator of granularity than is weight.

Other parallel logic programming architectures, such as independent-AND parallelism, can use the same techniques described in this paper in the context of stream-AND parallelism. We expect that non-stream-based architectures will also facilitate more accurate analysis of granularity based on the "unification strength" of procedure arguments.

6 Acknowledgements

As far as I know, the idea of compile-time granularity analysis was born in a discussion with J. Conery in

February 1988. This research was conducted at the Institute for New Generation Computer Technology (ICOT) under the auspicies of the NSF-ICOT Visitors Program. I thank M. Sato who wrote the KL1 system on the Sequent and helped debug the new schedulers.

REFERENCES

[1] S. K. Debray and M. Hermenegildo. personal communication, 1988.

[2] A. Goto. Parallel Inference Machine Research in FGCS Project. In *Proceedings of the First Japan-U.S. AI Symposium*, pages 21–36, December 1987.

[3] M. V. Hermenegildo. Relating Goal Scheduling, Precedence, and Memory Management in AND-Parallel Execution of Logic Programs. In *Fourth International Conference on Logic Programming*, pages 556–575. University of Melbourne, MIT Press, Cambridge MA, May 1987.

[4] N. Ichiyoshi. personal communication, April 1988.

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

[6] A. Okumura and Y. Matsumoto. Parallel Programming with Layered Streams. In *Symposium on Logic Programming*, pages 224–233. IEEE Computer Society, August 1987.

[7] M. Sato and et al. KL1 Execution Model for PIM Cluster with Shared Memory. In *Fourth International Conference on Logic Programming*, pages 338–355. University of Melbourne, MIT Press, Cambridge MA, May 1987.

[8] M. Sato and A. Goto. Evaluation of the KL1 Parallel System on a Shared Memory Multiprocessor. In *Proceedings of Working Conference on Parallel Processing*. IFIP, Pisa, April 1988.

[9] K. Taki. Measurements and Evaluation for the Multi-PSI/V1 System. In *France-Japan Artificial Intelligence and Computer Science Symposium*, pages 359–384, Cannes, November 1987.

[10] E. Tick. Performance of Parallel Logic Programming Architectures. Technical report, ICOT, 1-4-28 Mita, Minato-ku Tokyo 108, Japan, September 1988.

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

[12] K. Ueda. Making Exhaustive Search Programs Deterministic: Part II. In *Fourth International Conference on Logic Programming*, pages 356–375. University of Melbourne, MIT Press, Cambridge MA, May 1987.

[13] D. H. D. Warren. The SRI Model for OR-Parallel Execution of Prolog—Abstract Design and Implementation. In *Symposium on Logic Programming*, pages 92–102. IEEE Computer Society, August 1987.