Parallel Architectures and Compilation Techniques (A-50)
M. Cosnard, G.R. Gao and G.M. Silberman (Editors)
Elsevier Science B.V. (North-Holland)

A Fleng Compiler for PIE64

Hidemoto NAKADA, Takuya ARAKI, Hanpei KOIKE, Hidehiko TANAKA

H.Tanaka Lab., Dept. of Electrical Engineering, Faculty of Engineering,
University of Tokyo
7-3-1, Hongo, Bunkyou-ku, Tokyo 113, Japan
Email: {nakada,araki,koike,tanaka}@mtl.t.u-tokyo.ac.jp

Abstract: The programming language fleng is designed for highly-parallel execution of non-uniform problems. We cannot expect data concurrency from these problems. Therefore, fleng makes use of control concurrency which is derived from data dependency. Fleng execution efficiency depends on load distribution and scheduling.

We implemented a fleng system on the parallel inference engine PIE64. Our compiler performs static load partitioning and scheduling. Static load partitioning improves concurrency and reduces remote memory references. Static scheduling reduces synchronization costs arranging process execution order. These static optimizations require data-flow analysis. In this paper, we describe a data-flow analysis method in our system.

Keyword Codes: D.3.m
Keywords: Programming Languages, Miscellaneous

1 Introduction

In general-purpose highly-parallel systems, it is required to execute not only uniform problems, but also non-uniform problems, such as symbol processing. There has been many work for the uniform problems: for example, vectorizing and parallelizing compilers are available these days. These systems make use of data concurrency; however we cannot expect a lot of data concurrency from non-uniform problems. Therefore, these known techniques are inapplicable to non-uniform problems.

Fleng is a committed-choice language which can extract control concurrency from any problems. So, it is suitable for the highly parallel execution of non-uniform computation.

Efficiency of fleng execution depends on good methods for load distribution and scheduling. We describe our fleng compiler system for the parallel inference engine PIE64 focusing on static load partitioning and scheduling.

Load distribution itself is not difficult. However, naive distribution causes a lot of remote memory references and prevents efficient execution. The purpose of static load partitioning is to improve memory reference locality as long as maximum concurrency is maintained.

Hidaka [2] described static load partitioning based on execution profile. However, that method had two shortcomings: (1) it required too much time for analysis, and (2) it was

restricted by the initial input for the profiler. Here we show a method based on data-flow analysis, which separates goals into several units along with the data flow.

The purpose of the static scheduling is to decrease the number of synchronization called suspension. Suspension arises from inappropriate execution order of goals. Therefore, suspension can be avoided to a certain extent by rearranging the order. We arrange goals according to a result of data-flow analysis.

Fleng and PIE64 are briefly described in section 2. Section 3 gives an overview of the whole system. The compiler and its static load partitioning and scheduling methods are described in section 4. Finally, we describe the current status and future plans in section 5.

2 Fleng and PIE64

2.1 Committed-Choice Language fleng

Fleng is a member of a logic-based language family called Committed-Choice Languages. This family is a descendant of concurrent logic languages[3]: GHC and KL1[1] are well known members of this family.

A fleng program is a set of predicates. A predicate consists of a set of clauses. Clause is the minimum fragments of the program. A clause consists of a **head** and **bodies**. We separate head and bodies by ":-".

$$Head :- Body1, Body2, Body3.$$

Basically, a clause is a description of a rewriting rule, and the head represents the pattern which the clause can rewrite.

One of the most notable features of fleng is a single assignment variable. Variables have only two states: at creation time variables are **unbound** and by **binding** they change to a second state called **bound**. Binding is similar to assignment, but once bound to one datum, the variables never revert to unbound, and cannot be bound to other datum. A second and any subsequent binding attempts will have no effect.

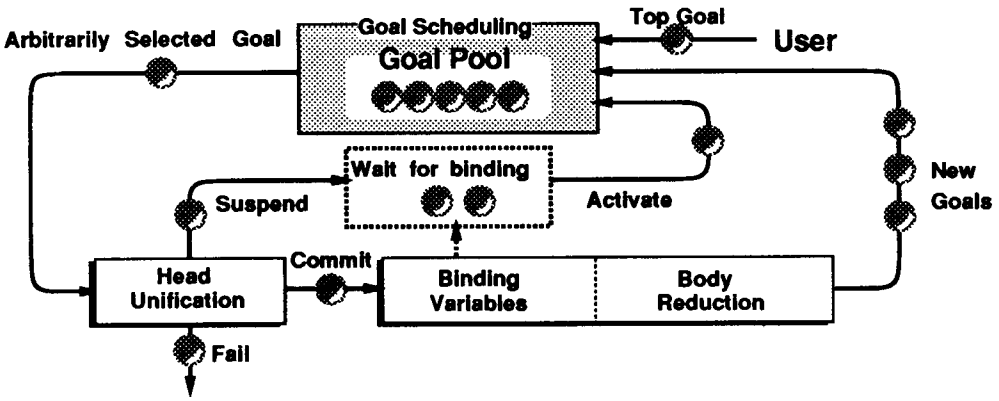

Figure 1: fleng execution model

Figure 1 shows a diagram of fleng execution. The basic execution unit of fleng is called a **goal**. While a fleng program is executed, there are many goals in the goal pool. Roughly speaking, the number of the goals in the pool is regarded as a measure of parallelism at the time. Fleng execution can be decomposed into three steps; (1) arbitrarily select a goal

from goal pool, and then, (2) rewrite the goal to other goals, at last, (3) return new goals to the goal pool. The rewriting is called **reduction**, and is done by the rule described by a clause which has the same name as the goal. A user starts execution by placing a goal, called the **top goal**, into the goal pool. When the goal pool becomes empty, execution stops.

To rewrite one goal, it is necessary that arguments of the goal are sufficiently bound to match with some of the clauses. If this condition is not met, it is impossible to rewrite the goal at that time. To wait for the appropriate condition, the fleng system **suspends** the goal according to the variable which is needed for the condition. This means that the goal is removed from the goal pool and, until the variable is bound, the goal will never try to be rewritten again. When the variable is bound, the system puts the goal into the goal pool again. This operation is called **activation**. Because of this synchronization system, no matter how we select a goal or when we rewrite it, it is guaranteed that result of a program execution is not affected by goal execution order.

In fleng, synchronization is expressed as a binding to variable and pattern match in clause head. This implicit synchronization enables fine-grained highly-parallel execution.

2.2 Platform for fleng : PIE 64

PIE64 is designed for fleng, and has dedicated hardware to reduce the costs for communication and synchronization, and to support load balancing. PIE64 also has hardware which supports parallel management.

Overview of PIE64 PIE64 consists of 64 processing elements called IU(Inference Unit) and two interconnection networks. The most notable feature of each processing element of PIE64 is to have three kind of processors. We divide parallel execution into three parts; computation, communication/synchronization, and parallel management, and assign three kind of processors these roles.

To reduce the cost of remote communication, PIE64 adopts latency-oriented interconnection networks, dedicated communication processors, and computing processors with a multi-context facility to hide latency. The communication processors also have a facility to support synchronization directly, to reduce the cost of synchronization. To support the load distribution, the interconnection network has a facility called automatic load balancing. With this facility, all IU can have access to the least load level, and the least loaded IU.

IU: Inference Unit An IU has three kinds of tightly connected processors: UNIRED (Unifier / Reducer) for fleng execution, NIP (Network Interface Processor) for communication and synchronization, and MP(Management Processor) for management.

UNIRED is a dedicated processor to fine-grained symbol processing[4]. It is designed with tag architecture and a dedicated instruction set. It also has several special features for organizing of parallel machines. The pipeline of UNIRED is shared by four contexts to hide remote reference latency.

NIP has two roles; communication and synchronization. The memory of PIE64 is distributed in each IU, but has one global address space. NIPs organize these distributed memory into a distributed shared memory. NIP also supports synchronization using single-assignment variable. Synchronization among IUs has to include some communication, so it is inevitable that one processor is in charge of these two roles.

MP is dedicated to parallel management. We use a general purpose SPARC processor, as MP.

3 Fleng system on PIE64

In this section, we review requirements of the system, and then show a sketch of our system.

3.1 Load Distribution and Scheduling

Load distribution and scheduling are important problems for implementing highly-parallel language systems. The optimum solution to these problems depends on the run-time situation. However, it is very difficult to predict perfectly the run-time situation of a non-uniform application. Therefore, it is probably impossible to obtain the optimum load distribution and scheduling using only a compiler.

Load Distribution Requirements for load distribution are as follows:

- To extract concurrency,
- To balance load,
- To reduce communication.

First, sufficiently high concurrency has to be extracted to fill all the IUs with some goals. Second, load, i.e. goals and data, have to be balanced among IUs in order to avoid overload on particular IUs. Third, the goals and data have to be allocated in such a way as to reduce the communication between IUs.

Scheduling An important requirement for scheduling is to minimize synchronization cost. Synchronization cost in PIE64 is low, because PIE64 hardware supports synchronization. However, some cost still remains. Suspension causes context changes on UNIRED and MP, and their costs also cannot be neglected.

Suspension arises because of inappropriate execution order of goals. Assume that two goals, a producer and a consumer, share one variable. The producer will bind some value to the variable, and the consumer requires the value for reduction. If the consumer is scheduled before the producer, the consumer suspends. However, if the producer is scheduled first, neither of them suspends. Therefore, scheduling by considering data dependency reduces suspension.

3.2 Overview of fleng system

To cope with load distribution and scheduling, we have adopted combination of static optimization and dynamic control.

The fleng system for PIE64 consists of a compiler system and a run-time system. On PIE64, compiled code on UNIREDs and a **run-time kernel** on MPs cooperate to execute fleng programs.

Compiled code on UNIRED can concentrate on computation, since MP and NIPs take charge of other bothersome tasks. UNIRED only receives a goal from MP, and reduces it, and sends back new goals to MP. If a goal cannot be reduced, UNIRED simply quits

execution. MP and NIPs are responsible for suspending the goal. Whenever remote access is required, UNIRED automatically sends a command to NIP.

The principal role of the run-time kernel is goal management. The goal management is to fill up all UNIREDs with goals. The goal pool is distributed across all IUs in order to avoid a bottle neck. Each MP manages a goal pool on the IU, i.e. it supplies UNIRED goal, gets new goals from UNIRED, and when the need arises, sends goals to other IUs or recieves goals from other IUs.

Load distribution on PIE64 Load distribution on PIE64 is handled at three stages as shown below.

1 Static load partitioning by compiler,
2 Dynamic load distribution by run-time kernel,
3 Dynamic load balancing by interconnection networks.

In the first stage, the compiler partitions loads; i.e. newly created goals and data. The role of the first stage is to enhance memory reference locality as long as all possible concurrency is maintained. The compiler detects data dependency between goals to allocate related goals to the same IU. In the next section, we will describe the details of this partitioning.

In the second stage, using run-time information, the run-time kernel decides whether the load should be distributed. Under a situation that all other IUs are sufficiently loaded, distributing load is not only useless, but can worsen reference locality. Therefore, under such a situation, the run-time kernel keeps all newly created goals inside the IU to suppress excessive concurrency.

At last, in the third stage, the automatic load balancing facility of interconnection networks is used to send goals to the least loaded IU.

4 Compiler and Static Optimization

4.1 Overview of Compiler System

Figure 2 shows a brief diagram of the fleng compiler system. This system consists of a mode analyzer, an optimizer and the compiler. All of them are written in fleng themselves.

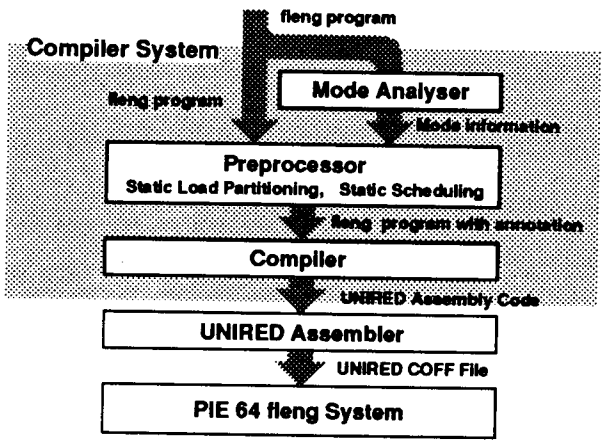

Figure 2: fleng compiler system

The optimizer does load partitioning and scheduling according to information obtained by the mode analyzer. Decisions made by the optimizer are added to the program as annotations. The compiler generates UNIRED assembly code according to the annotations. We separate the optimizer and the compiler to evaluate various kinds of optimization policy without any modification to the compiler.

4.2 Annotations

We use annotations to inform the compiler about the decisions determined by the optimizer.

The syntax of the annotation produced by the optimizer is as follows, where *Item* is a variable, a structured term or a body goal.

Item `@[` *annotation1, annotation2,....* `]`

The result of load partitioning is specified with the following annotation:

- `local`
  This annotation means that the goal/data should be allocated in the local IU.
- `on(` *label* `)`
  This annotation is used in the head part of clauses. The IU where the data resides is tagged as *label.*
- `to(` *label* `)`
  This annotation means that the goal/data should be allocated in the IU tagged as *label.*
- `any(` *label* `)`
  This annotation means that the goal/data should be allocated in the least loaded IU. The IU where the goal/data is allocated is tagged as *label.*

The result of static scheduling is specified with the following annotation:

- `sequence(` *number* `)`
  This annotation specifies an execution order of the goals which are allocated in the same IU.

4.3 Static load partitioning and Static scheduling

As mentioned in section 3, the role of static load partitioning is to enhance memory reference locality as long as all concurrency is maintained. To achieve this objective, data dependency information is required.

The point of static scheduling is to arrange goal execution order. This also requires data dependency information. In both cases, we get the information by data-flow analysis.

4.3.1 Simple mode analysis

Our data-flow analysis consists of two phases; mode analysis and data-dependency analysis. In the first phase we determine the mode of predicates. In the second phase, using the mode information, we analyze the data dependency in each clause.

Although there are several work on mode system for the committed-choice languages, they specify only input/output mode. The second phase requires data-dependency information, we create a new mode system which includes data-dependency information.

Predicate mode is represented by a list of argument modes. Our mode system uses the following 5 argument modes.

++ **strong input**: required immediately
+ **input**: required but not immediately
−− **strong output**: guaranteed to be bound
− **output**: not guaranteed to be bound
? **unknown**

For example, consider the following program fragment:

```
foo(a, B):- B = b.
```

This predicate requires that the first argument is bound, and guarantees that the second argument is bound after the execution. As a result, the mode of this predicate is specified as `[++, --]`.

To get full mode information, a global analysis is required. However, local analysis will be enough to get the data-dependency information. Therefore, we decided not to do global analysis.

The mode of predicates is determined by synthesizing mode of clauses. Clause mode is also specified as a list of argument modes. The mode of clause is determined by the mode of each arguments. The argument mode is determined by the following rules:

- In the head, if the argument is not a variable, the argument mode is specified as strong input.
- In the head, if the argument is a variable, and there are some goals which binds the variable to some non variable object, then the argument mode is specified as strong output.

The actual rules are somewhat more complicated, but omitted here.

The n-th argument mode of a predicate is synthesized from the n-th argument modes of clauses. Each argument mode is determined by the following rules:

- If all the modes of clauses are same, the mode is taken as the argument mode.
- If strong input and input conflict, input is selected.
- If strong output and output conflict, output is selected.
- If unknown and any other mode conflict, other mode is selected.
- If input and output conflict, unknown is selected.

4.3.2 Data-flow graph

From the predicate mode, we can get a data flow directed graph, using variables and goals as nodes, and data dependency as arcs. The strong-input mode indicates that a goal depends on an argument which have the mode. The strong-output mode indicates that an argument which have the mode depends on a goal. A data-flow graph can be obtained by the following steps:

1. Treat variables as data nodes.
2. Treat body goals as goal nodes.
3. If a body goal has a strong input mode on a variable, make arcs from the data node to the goal node.
4. If a body goal has a strong output mode on a variable, put arcs from the goal node to the data node which represent the variable.

4.3.3 Load partitioning and scheduling

Load partitioning and scheduling problems can be resolved into the data-flow graphs partitioning. For any walk in the graph, any two goals on the walk can not be reduced simultaneously. In other word, for any two goals, no graph walk includes both of them, they can be reduced simultaneously. Therefore, the load-partitioning problem can be solved by partitioning of data-flow graph.

The scheduling problem can also be solved by graph partitioning. In the data-flow graph, upstream goal should be reduced in advance. Therefore scheduling can be done by arranging the goals according to the graph walk.

We partition the data-flow graph with the following steps:

1. Select one of the longest walks as the target arbitrarily.
2. Remove all the goal/data nodes and arcs which are included in the target walk, and allocate them to one processor;
3. By the previous operation, if some data nodes are isolated from the graph, allocate the data node and arc to the same processor as 2;
4. By operation 2, If some goal nodes are isolated, allocate them on another processor;
5. Repeat these operation until the graph becomes empty. If the graph is separated into plural graphs, partition each graph.

4.3.4 Example

We show an application of our method to a program fragment as an example. The following clause is a part of an n-queen program. The predicate mode of `add`, `sub`, `equal`, and `chk` are `[++,++,--]`,`[++,++,--]`, `[++,++,--]` and `[++,++,?,?,?,?,?,?]`, respectively.

```
check(P, D, L, [Q|Lp0], Lp, A0, A):-
    add(Q, D, Sum @ [any(1)])@ [to(1), sequence(1)],
    equal(Sum, P, R1 @ [to(1)])@ [to(1), sequence(2)],
    sub(Q, D, Dif @ [any(2)])@ [to(2), sequence(1)],
    equal(Dif, P, R2 @ [to(2)])@ [to(2), sequence(2)],
    chk(R1, R2, P, D, L, Lp0, Lp, A0, A) @ [to(2), sequence(3)].
```

Figure 3 shows a diagram of data flow and load partitioning specified by the annotations in the above list. We can see that the graph partitioning is done according to the data flow.

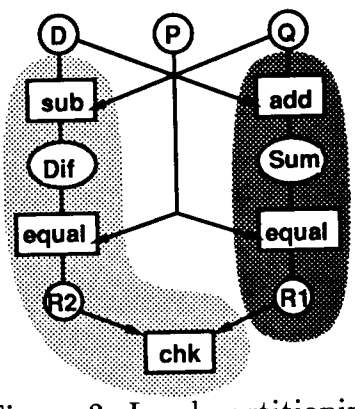

Figure 3: Load partitioning

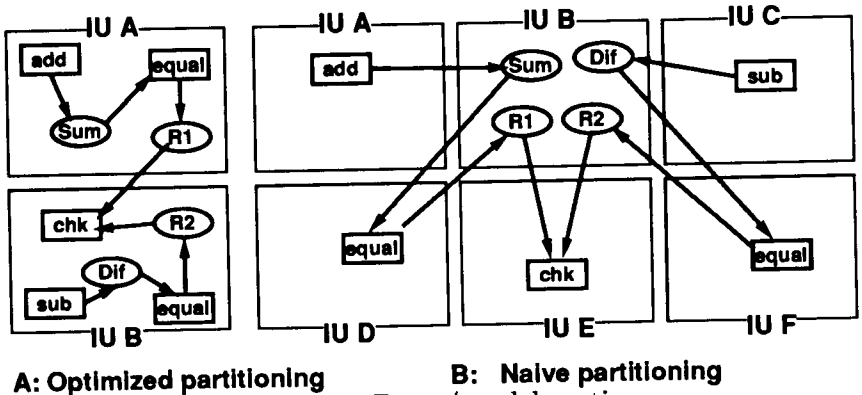

Figure 4: Data/goal location

Figure 4:A shows the resulting allocation. The large square represent each IU, and arcs between goals and data represent data references. Figure 4:B shows the worst allocation caused by naive load partitioning; it separates all the goals and locates all the data on the local IU. By optimal partitioning, only one remote data reference takes place. In contrast, all the memory accesses are remote with naive partitioning. Note that concurrency is not reduced by the optimized partition, even though only two IUs are used.

4.4 Preliminary Evaluation

We have done preliminary evaluation to gain some understanding about the relation between program concurrency and the effect of our dynamic/static distribution method.

We customized a fleng interpreter on PIE64 for evaluation. The interpreter distributes goals and data as specified by the annotations. It shares the run-time kernel with the compiler system, and has the dynamic load-distribution facility mentioned above in Section 3. The facility can be disabled to evaluate the facility itself.

For the evaluation, we used a well-known program "primes", which finds all prime numbers less than 200. This program has relatively low concurrency; 15 - 20 on average. We executed several of "primes" simultaneously, to get variation in concurrency.

Among the "primes", there are no communication, therefore, the absolute value of locality will be different between the result of this experiment and real applications. However, the tendency of the locality against the program concurrency will be same.

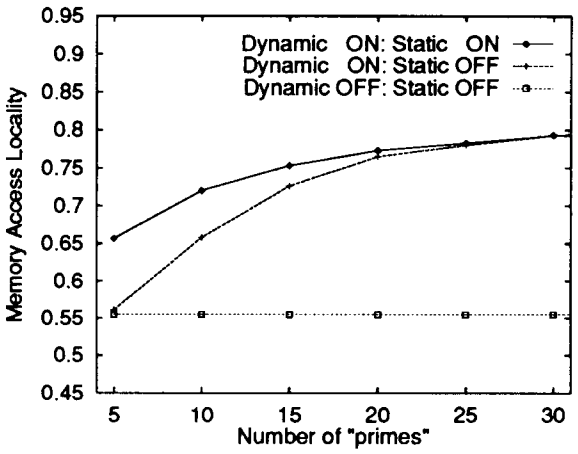

Figure 5: Locality of primes

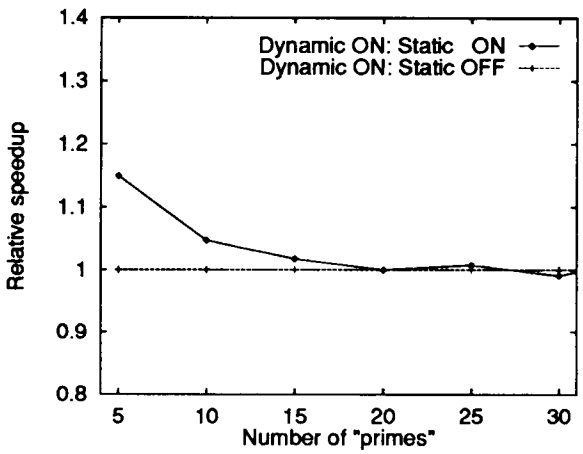

Figure 6: Relative speed up

Figure 5 shows the memory reference locality against the number of "primes". The lowest graph shows the most naive method, i.e. no dynamic load distribution and no static load partitioning. It always shows same low locality. The middle graph shows the result of the dynamic load-distribution. When the concurrency is high, it shows enough locality, however, the method shows no improvement in low concurrency area. The highest graph shows the result of the combination of the dynamic load distribution and the static load partioning. The graph keeps highest locality among three.

The higher the concurrency becomes, the smaller the difference between the two graphs. When all other IUs are loaded, the run-time kernel does not distribute any loads to suppress excessive concurrency. Therefore, when the concurrency is high, the effect of the dynamic load distribution dominates, and it hides the effect of static partitioning.

Figure 6 shows the relative speed up of statically load partitioned programs compared with the naive one, against the number of "primes". The dynamic load-distribution

facility is enabled in both case. Note that the reduction speed of interpreter is low, so the remote access penalty appears to be relatively low. As implied by the memory reference locality, the higher the concurrency becomes, the smaller the relative speed up.

From the results described above, we conclude that; (1) the dynamic load distribution is not enough in low-concurrency area, (2) the static load-partitioning is effective to fill up the weak point of the dynamic load distribution.

5 Conclusion and future work

We have described a fleng compiler and its static load partitioning and scheduling based on data-flow analysis. Static load partitioning is effective, especially in the low concurrency area.

For future work, the following are considered to be important:

- Using profiling data in static scheduling,
- Adopting more complicated static analysis,
- Evaluating our whole system.

Here, we only use static data. However, for selecting targets from the several longest walks, simple profiling data will be useful.

Our mode analysis is relatively naive and simple, and it restricts static optimization. We are planning to establish more complicated mode analysis. With that analysis, other static optimization is possible; for example, grain combination.

Finally, evaluation of our whole system is required, including the run-time kernel.

Acknowledgments

This work has been supported by Grant-in-Aid for Scientific Research (No.62065002, No.03555071, No.03003891) from the Ministry of Education, Science and Culture.

We are thankful to Professor Randy Goebel for editing an earlier draft of this paper.

References

[1] Takashi CHIKAYAMA. Operating system PIMOS and kernel language KL1. In *Proc. of International Conference on FIFTH GENERATION COMPUTER SYSTEMS 1992*, pages 73–88, 1992.

[2] Yasuo HIDAKA, Hanpei KOIKE, Jun'ichi TATEMURA, and Hidehiko TANAKA. A static load partitioning method based on execution profile for committed choice languages. In *Proc. of ILPS*, 1991.

[3] Ehud SHAPIRO, editor. *Concurrent Prolog.* The MIT Press, 1987.

[4] Kentaro SHIMADA, Hanpei KOIKE, and Hidehiko TANAKA. UNIRED II: The high performance inference processor for the parallel inference machine PIE64. In *Proc. of Int. Conf. of Fifth Generation Computer Systems*, pages 715–722, 1992.