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

OVERVIEW OF THE PARALLEL INFERENCE MACHINE OPERATING SYSTEM (PIMOS)

Takashi Chikayama Hiroyuki Sato
Institute for New Generation Computer Technology
4-28, Mita 1-chome, Minato-ku,
Tokyo 108, Japan

Toshihiko Miyazaki
Oki Electric Industry Corporation
11-22, Shibaura 4-chome, Minato-ku,
Tokyo, 108, Japan

ABSTRACT

The parallel inference machines are being developed in the Japanese FGCS project to provide the computational power required for constructing high performance knowledge information processing systems. To fully exploit the power of parallel inference machines, an operating system tuned to control highly parallel programs effectively is inevitable. The parallel inference machine operating system, PIMOS, is designed for this purpose. This paper describes an overview of the design of the PIMOS.

The description language of the PIMOS, KL1, is based on a concurrent logic programming language, Flat GHC. To obtain the functionality required for writing a complicated system such as an operating system, the KL1 language made numerous extensions to the original GHC language, mainly for efficient meta-control.

Based on the features provided by the KL1 language, the PIMOS is designed to be an efficient, robust and flexible operating system tuned to the parallel inference systems. Through its development, implementing an operating system in a concurrent logic programming language has been proved to be not only feasible but also advantageous.

1 INTRODUCTION

1.1 Objective

The parallel inference machines, PIM's (Goto *et al.* 1988), are being developed in the Japanese FGCS project to provide the computational power required for high performance knowledge information processing systems. A prototype parallel inference machine, Multi-PSI (Takeda *et al.* 1988) has also been developed to promote parallel software research and development. These systems consist of multiple (up to around 1000) processors for attaining the required processing power.

To fully exploit the power of such parallel inference machines, an operating system tuned to control highly parallel programs effectively is inevitable. The system should also be user-friendly and robust enough to be used practically and extensively in parallel software research. The parallel inference machine operating system, PIMOS, is designed to fulfill this requirement.

1.2 Related Works

The possibility and advantages of writing a complete operating system in a concurrent logic programming language are suggested by Shapiro (1984). Based on this principle but with much improvements in various aspects, several experimental systems such as the Logix system (Hircsh *et al.* 1987) and the Parlog Programming System (PPS) (Foster 1987) are actually implemented.

The PIMOS resembles PPS in many aspects. This resemblance is partly due to the resemblance of the implementation languages (KL1 and Parlog) and partly due to intimate cooperation of two research groups.

A notable difference between the PIMOS and other above-mentioned systems is in the underlying language implementations. The PIMOS is designed for hardware specially devised for parallel inference systems with very high performance, while other systems are built upon commercially available software and hardware. This affects the execution efficiency of various language primitives differently, changing design trade-offs considerably.

1.3 Characteristics of the Hardware Systems

The hardware systems for which the PIMOS is designed have the following characteristics in common.

Stand-Alone Systems: The parallel inference machines are designed to be stand-alone systems; not as back-end processors of established host systems.

Multiple Processors: The parallel inference machines have many processors that can execute different programs in parallel. All processors have the same functions; any processor can take any part of the system. Job allocation is left to the software.

Loosely Coupled Processors: In the Multi-PSI system, all the processors are connected loosely via a specially devised communication network. In certain PIM systems, several processors are connected tightly, sharing a common bus and memory, forming

a *cluster*. Clusters, however, are interconnected via a communication network. As the inference mechanism itself is highly optimized, communication between processors (or clusters) through the network is relatively costly, and the software must take more care of keeping locality of computation. Especially, the highest cost is in the fixed per communication overhead.

Changing Parameters: We do not have much experience with highly parallel inference systems yet. Although all the parallel inference machines are based on the same design principles, various parameters of the systems may differ depending on our knowledge on such systems available at the time of their design. Also, even for one model, parameters may change in time as the system is gradually tuned up. The same applies to the implementation technique of the KL1 language. Such parameters may considerably affect trade-offs in the software design.

The PIMOS is designed keeping these characteristics of the hardware in mind.

1.4 Requirements

The following items are required for an operating system for systems built upon the hardware with the characteristics described above.

Robustness: As the PIMOS is a stand-alone system, the robustness of the system is more important than in systems based and depending upon another established system.

Parallelism: The ultimate objective of the PIMOS is, as stated above, to provide features that fully exploits the power of parallel inference hardware. Various computations required in such an operating system should also be executed in parallel. Otherwise, the operating system will be the bottleneck of the whole system.

Low Communication Frequency: As the processors or clusters are loosely connected, communication between them are much more costly compared with communication within one processor. Thus, frequency of communication between processors should be kept as low as possible.

Flexibility: As the hardware parameters are expected to change, the system should have enough flexibility to be tuned to the given parameters. When tuning by changing parameters of the operating system becomes insufficient, non-trivial re-design of the system may be required. Thus, a system is desirable on which improvement of the system itself is easy. Features enabling construction of so-called *virtual machine operating systems* are required from this viewpoint.

1.5 Organization of the Paper

The rest of this paper is organized as follows.

Section 2 describes the implementation language of the system, KL1. Many of the features of the operating system PIMOS is based upon the primitives of the KL1 language provided as its meta-level control features. Thus the design of the KL1 language, especially extensions made to its base language GHC, should be considered to be a part of the design of the PIMOS.

Section 3 describes how physical input and output devices are modeled in KL1, what kind of logical interface is provided to the user, and how they are realized.

Section 4 describes how executable programs of the KL1 language are stored and used for execution in the PIMOS.

Section 5 describes how various resources are controlled in the PIMOS.

Section 6 describes how the user programs and the PIMOS can communicate to each other, and how the communication is made in a fail-safe way to protect the PIMOS from accidental or intentional errors of user programs.

Section 7 describes the environment prepared for the development of the PIMOS and other parallel application software.

Finally, in the last section 8, a conclusion and plans for future research and development directions are stated.

2 THE KL1 LANGUAGE

The implementation language of the PIMOS is called KL1, the common kernel language for parallel inference systems in the FGCS project, based on the GHC language (Ueda 1986). GHC is a concurrent logic programming language akin to Concurrent Prolog (Shapiro 1983) or Parlog (Clark and Gregory 1986).

The merit of using a concurrent logic programming language is in its implicit concurrency and synchronization feature. Without explicitly specifying in the program, concurrency of the program is exploited and data-flow synchronization is made automatically in and under the language implementation level. Especially advantageous is the implicit data-flow synchronization mechanism which eliminates almost all the synchronization errors. In a procedural language, required data-flow synchronization must be converted to control-flow synchronization by the system program, which is one of the largest sources of programming errors in operating systems.

KL1 is actually based on a subset of GHC called *Flat* GHC, or FGHC in short. The difference of *flat* version of GHC and its *full* version is that only unification and calls to certain built-in predicates are allowed in the guard

part of a clause. This makes efficient implementation considerably easier, without losing essential descriptive power of the language.

However, the GHC language itself does not have enough power for efficient implementation of operating systems or application programs that require sophisticated control mechanism. Thus, several extensions are made to the language, mainly for enabling *meta-level* execution control. This section describes why such extensions are required, what sort of extensions are made, and how they are supposed to be used.

2.1 Requirements

For describing large scale programs requiring complicated execution control, a reasonable structure should be introduced to the program. One of the reasons of the requirement of such a structure is to keep each level of the structure small enough to be comprehended easily at a time. Another reason is to map the structure of the problem directly to the the structure of the program, which also helps easier comprehension.

One way to introduce such a structure to programs is by dividing the program into modules statically. Development of the languages such as Vulcan (Kahn *et al.* 1986) or *A'UM* (Yoshida and Chikayama 1988) are to build modular programming languages based on the object-oriented notion upon concurrent logic programming languages. This approach is known to be effective for solving many problems and the KL1 language does provide a simple modular program structure also, but unfortunately it is not enough by itself for describing an operating system.

In operating systems, *not* all objects are created equal. The operating system should be able to control the execution of the application programs, and the reverse should not hold. The program that controls the execution of a program is called its *meta-program*; an operating system is a meta-program of the application programs. This meta/object structure is not a structure straightforwardly expressed in modular programming languages.

The simplest and probably the most elegant way to implement the meta-programming feature may be providing an interpreter of a programming language (Shapiro 1984). If the operating system should interpret the application programs under its supervision, any kind of meta-control could have been implemented easily. Such an implementation, however, has an obvious drawback in execution efficiency.

The same sort of meta-control feature required in operating systems is also required for certain kinds of application programs. For example, the command interpreter shell is the meta-program of programs run under it. Programs controlling several solvers with different algorithms for the same problem is the meta-program of the solvers. The operating system is no more than an instance of programs requiring meta/object program structures.

Thus, the layers of meta-control can be nested arbitrarily many times. If the execution efficiency should be reduced to $1/r$ by using the interpreter scheme, the efficiency of a program within n levels of meta-control layers will be r^n times as slow as when it is executed directly by the machine. The partial evaluation technique can solve the problem partly, lowering the overhead of interpretation considerably. Nevertheless, it can only lower the constant factor r and cannot (with currently available technology, at least) make it very close to 1 either when powerful meta-control is required (Hircsh *et al.* 1987). Thus, to encourage meta-level control, a mechanism allowing object-level and meta-level programs to run on the same basis with the same efficiency is required.

The following features should be available in such a meta-programming mechanism.

Preventing Propagation of Failure: In FGHC, all the goals in the system form one large logical conjunction. Thus, failure of one goal in the system means the failure of the whole system. If the meta-level program and supervised object-level programs are to be run this way, failure in an object-level program will cause failure of the whole system including the meta-level program, which is never acceptable. Thus, propagation of the failure should be limited somehow to the object-level, to prevent the failure of the meta-level program.

Meta-Control: The meta-level program should be able to control the execution of the object-level programs. For example, a user should be able to stop his job from the command interpreter shell, when one of his jobs went into a meaningless infinite iteration.

Monitoring: The meta-level program should be notified of exceptional events (arithmetical overflow, for example) raised in the object-level and be able to determine what to do with such events. In general, the meta-level program should be able to monitor the execution of the object-level programs at any time, to be able to control object-level programs based on the monitored information.

2.2 The "Shōen" Feature

For introducing the meta-programming feature, an appropriate program structure should be introduced to the FGHC language to distinguish the meta-level and the object-level. The feature of *Shōen*[1], similar to the meta-call primitive in Parlog (Clark and Gregory 1984), is introduced for this purpose as a language primitive.

[1] The word "shōen" (or "荘園") is a Japanese word that means "manor" in English.

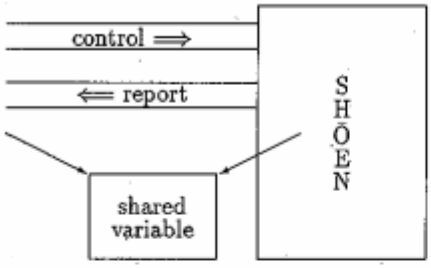


Figure 1: Shōen

The shōen mechanism can be considered to be an interpreter of the KL1 language. Although it is not actually written in KL1 but is implemented by lower level primitives, its semantics is designed so as to preserve the nature of a meta-level interpreter.

2.2.1 Creation of Shōen

A shōen is created by the following primitive.

execute(Goal, Control, Report)

This can be considered to be a call of the top-level predicate of the interpreter. Here, each argument means the following.

Goal: The goal to be executed in the shōen.[2]

Control: A stream from outside of the shōen to the shōen interpreter, through which commands for controlling the execution of the interpreter are sent.

Report: A stream from the shōen interpreter to outside of the shōen, through which various messages are sent from the shōen interpreter to report status of the computation.

Here, the word *stream* actually means a list structure used for stream-like communication between processes (Shapiro and Takeuchi 1983). Each argument will be described further in detail below.[3]

The goals derived from the original goal given to a shōen on its creation form a logical conjunction independent of the goals outside the shōen. Once a shōen is created, its execution is controlled through the *control* stream. Status of the execution of the shōen is notified from the *report* stream. Thus, as far as execution control is concerned, a shōen is a blackbox with one input stream and one output stream.

[2]In the actual implementation, it is represented by 2 arguments: a pointer to the executable code and an argument vector. See 2.6.2 for this design choice.

[3]Actually, the *execute* primitive has several more arguments, which also will be described below.

The original goal given at the creation of a shōen, however, can have variables shared with goals outside of shōen as its arguments. Goals in the shōen can instantiate such variables, thus sending information out; goals out of the shōen may also instantiate such variables, thus sending information into the shōen. In terms of virtual interpreter assumption, this means that the variables in the interpreted program is represented by the variables of the language in which the interpreter is written, as is usually the case with Prolog interpreters.[4]

2.2.2 Controlling the Execution

Execution of a shōen virtual interpreter can be controlled by sending the following messages to the *control* stream of the shōen.

Start: Start (or restart) the execution of the shōen. After its creation, the execution of the shōen is suspended until this message is first received. The same message resumes the execution of the shōen suspended by a *stop* message described below.

Stop: Stop the execution of the shōen. The execution is suspended until a *start* message is received. To allow efficient implementation, the language allows arbitrarily long (but finite) delay until the execution is actually stopped.

Abort: Abort the execution of the shōen. The execution is aborted and (unlike in the case of the *stop* message) can never be resumed afterwards. Again, arbitrary finite delay is allowed here.

In addition to those listed above, commands for resource management are also sent through the same stream (see 2.5).

2.2.3 Reporting Status

The status of the shōen are reported through the *report* stream by the following messages.

Started: Reports the reception of a *start* command.

Stopped: Reports the reception of a *stop* command.

Aborted: Reports the reception of a *abort* command.

Terminated: Reports the termination of the execution. The termination takes place when all the goals in the shōen are reduced. Alternatively, the shōen may have been forced to terminate by an *abort* message.

[4]Using this efficient but simple mechanism, however, unexpected instantiation of variables in the object-level may cause failure in the meta-level. In the PIMOS, it is solved by the protection filter technique described in 6.4 with the help of the unification order rules described in 2.2.5.

The first three in the above list are used for deciding the order of commands reception and other internal events. For example, when a *terminated* report is made after an *abort* message is sent to a shōen, there may be two cases: when the execution is aborted by the command, and, when all goals in the shōen has been reduced successfully *before* receiving the *abort* message. These two cases can be distinguished by the order of the *aborted* and *terminated* messages in the report stream.

In addition to those listed above, messages to report exceptions and resource consumption status are also sent to the the report stream (see 2.3 and 2.5).

Note that *failure* is not included in the above list. In KL1, failure is treated as a kind of exception (similar to arithmetical overflow). The meta-level program decides whether to abort the execution (by sending an *abort* message) or try to make it recover from the failure (see 2.3).

2.2.4 Nested Shōens

To allow flexible meta-programming, shōens can nest by arbitrarily many levels.

As the semantic model of a shōen is a virtual interpreter, the semantics of nested shōens is the same as when an interpreter is interpreted by another interpreter. It may be naturally understood that suspending the outer shōen also suspends the inner shōen; stopping the outer shōen is stopping the interpreter that interpretively executes the inner interpreter. Resuming the outer shōen will also resume the inner shōen. Similarly, aborting the execution of one shōen also aborts the execution of its offspring shōens.

With this semantics, the meta-level program can supervise object-level programs without being aware of any lower meta/object layers.

2.2.5 Order of Unifications

No order between distinct unifications is defined in the original GHC language. For example, consider the following KL1 program.

```
p :- q(a).
q(X) :- X = b.
```

Naively considering, when the predicate p is called, the unification `X = b` in the predicate q will fail. However, The first clause is considered to be equivalent to the following clause in the definition of original GHC.

```
p :- X = a, q(X).
```

The order of the unification "`X = a`" and the invocation of the predicate q is not defined. Thus, the unification "`X = b`" may be executed prior to the unification "`X = a`" in the predicate p, which will fail if this is the case.

This theoretically clean semantics brings in a problem in protecting the meta-level using the shōen mechanism. The same failure may occur even when the predicate q is called wrapped up in a shōen construct as in the following clause.

```
p :- execute(q(a), ...).
```

To avoid the above and similar problems, the following order of unification is assumed in the KL1 language.

- When there are two or more occurrences of the *same* variable, for example, several occurrences of a variable X, in one clause, they are *unified* before the body goals are invoked.
- What are passed as arguments to a body goal are the arguments as written in the program, rather than variables which will be unified with the written arguments later.
- When a structure appears in the body part, its elements are initiated with the values written in the program, rather than variables which will be unified with the written ones later.

Fortunately, there seem to be no reasons for an optimized implementation to violate these rules.

2.3 Exception Handling

Exceptional events during the execution is reported to the *report* stream of one of the surrounding shōens.

2.3.1 Causes of Exceptions

Typical causes of exceptions are the following.

- When invalid arguments are given to a built-in predicates in the body part of a clause. For example, giving non-numerical arguments to arithmetical built-in predicates, giving arguments that cause arithmetical overflow,[5] giving an index value that is out of the range of the given structure, etc., fall into this category.
- When the guards of all the candidate clauses for a goal are known to fail.
- When an active unification[6] fails.

Built-in predicates appearing in the guard of a clause will never cause exceptions. Instead, when a built-in predicate, say addition, is given an invalid argument,

[5] Arguments which cause arithmetical overflow are considered to be *invalid* here.

[6] *Active* unification is one appearing in the body part of a clause, which can instantiate unbound variables. *Passive* unification appearing in the head or the guard part of a clause will never give values to variables.

say an atom, it simply fails rather than generating an exception. Built-in predicates appearing in the guard part are considered to be abbreviations of unification patterns. For example, consider the following clause.

```
p(X, Y) :- X > Y | q(X).
```

This is considered to be an abbreviation of the following infinitely many clauses.

```
p(1, 0) :- q(1).
p(2, 0) :- q(2).
...
p(2, 1) :- q(2).
p(3, 2) :- q(3).
...
```

2.3.2 Reporting Exceptions

When an exceptional event is found, a message as shown below is sent to the *report* stream of one of the shōens surrounding the goal that caused the exception.

exception(Info, Goal, NewGoal)

Each argument of the exception information has the following meaning.

Info: The reason of the exception.

Goal: Information on the goal which caused the exception.[7]

NewGoal: A variable to specify a goal that will be executed in place of the original goal that caused the exception.[8]

There may be any number of shōens surrounding the goal that caused the exception, but only one of them receives the exception message. Exceptions are classified into several categories and each category is associated with some *tag* (one word bit pattern). On the other hand, every shōen also has a *tag*, which is specified on its creation by an additional argument to the *execute* primitive. The exception is reported to the innermost shōen whose tag *matches* the tag of the exception. Two tags match when their bit-wise conjunction yields non-zero. Using this mechanism, one shōen monitor can handle only certain kinds of exceptions, leaving others handled by the monitors of outer shōens.

[7]In the actual implementation, it is given by two terms; a code pointer and an argument vector.

[8]Like the *goal* information, two variables for a code pointer and an argument vector are used in the actual implementation.

2.3.3 Recovering from an Exception

When an exception report is generated, the execution of the goal that caused the exception (a built-in predicate goal or a failed goal) is replaced by a new goal. That goal waits for the instantiation of *NewGoal* in the exception report message and, after its instantiation, executes it. As this new goal belongs to the same shōen as the original goal, the execution of the shōen will not terminate successfully before the *NewGoal* argument is instantiated.[9]

The semantics of the language can be partly customized by specifying an appropriate *NewGoal* in the shōen monitor program. The semantics of unification can be extended, for example, by giving user-defined unification routine as the *NewGoal* for a unification failure exception.

Even when an exception is reported to the report stream of a surrounding shōen, the execution of other goals in the shōen will *not* be suspended. Whether to stop the execution or not is determined by the monitor program of the shōen via the control stream of the shōen. In parallel implementation of the language, it is practically impossible in anyway to stop the execution immediately.

2.3.4 Deliberate Generation of Exceptions

An exception report can be intentionally generated using the following primitive.

raise(Info, Data, Tag)

Each argument has the following meaning.

Info: Any data identifying the exception. The generation of the exception is deferred until this argument is instantiated completely to a ground term.[10]

Data: Any data. This argument may be instantiated, uninstantiated or partly instantiated.

Tag: An integer to specify the tag of the exception, which, in turn, specifies the shōen whose monitor handles the exception.

When an object-level program sends some information to its meta-level, the part of the data that is inspected by the meta-level should be guaranteed to be instantiated. Otherwise, the object-level program may fail to instantiate it, causing the meta-level program wait for it forever. The argument *Info* is used for this kind of information.

On the other hand, the argument *Data* is used to pass data that are *not* inspected by the meta-level program.

[9]Abortion is possible at any time.

[10]In the actual implementation, the mechanism of deferring the exception report is implemented by a KL1 predicate. However, it is a language feature from the users' point of view.

They are usually passed directly to the goal that is executed in place of the goal *raise*, by including it in the term unified with the *NewGoal* argument of the exception report. As the substitute goal is executed in the object-level, the problem of deadlock in the meta-level will not appear.

A typical usage of this feature is for establishing a communication path from a user program to the PIM-OS, described in section 6.2.

2.3.5 Implicit Stream Argument

The exception mechanism of KL1 can be explained by assuming one additional implicit stream argument to each goal. This implicit argument is unified with [] when the clause has no body goals. When it has body goals, an implicit merger goal is inserted in the body which merges as many streams as the number of body goals to the implicit stream argument, and pass one merged-in stream to each of the body goals as their implicit stream arguments.

For example, a clause such as:

```
p(X,Z) :- q(X,Y), r(Y,Z).
```

is considered to represent a clause:

```
p(X,Z,S) :-
  q(X,Y,S1), r(Y,Z,S2), merge(S1,S2,S).
```

The same rule applies also to built-in predicates in the clause body. This implicit stream is virtually merged into the report stream of the shōen, through which exceptions are reported.

2.4 Priority Management

For specifying sophisticated problem solving strategy that can fully utilize the available computational resources in an effective manner, it is essential to introduce the notion of *priority* between *goals* that can be executed in parallel and between *clauses* that can be chosen non-deterministically.

2.4.1 Requirements

In the original GHC language, the execution order of two goals can be either of the following two.

- The order is not specified. The order is left to the implementation, and the implementation may sequentially execute one after the other, or may execute them in parallel.
- The order is determined by data dependency. One can be executed only after the other makes some data available.

These two may be enough as far as there is no limit in the available resource, because, in that case, everything that can run in terms of data dependency can really proceed. However, in an actual implementation where only limited resource must be fully utilized, the following strategy is often desirable.

- The two goals may be executed in parallel, as far as both can be.
- If there is not enough computational resource (processors, for example), execution of one should have priority to the other.

Consider, for example, the alpha-beta tree search algorithm. It is essential in the algorithm to search one branch thoroughly as early as possible, to utilize its result for pruning other branches. If the search should have been made in the breadth-first order, no pruning procedure would be possible.

When programmed in sequential programming languages, strict depth-first search order is specified. The same kind of strict sequentiality can also be specified in GHC using data dependency, in which case, however, the algorithm cannot make use of otherwise available idle processors.

To solve the problem, more flexible notion of *execution priority* is required, in addition to the strict ordering enforced by data dependency. When there are several computations ready to be executed in terms of data dependency but with different priorities, and the computational resource is available only for some of them, ones with higher priority will be executed first. The essential difference with the strict ordering is that all the computation may be tried in parallel if abundant resource is available.

Priority is not something to be obeyed strictly but merely a guideline suggested to the implementation to determine the execution order of goals. Thus, goals with lower priority *may* be executed even when there exist goals with higher priority ready to be executed. How much the priority is respected determines how good the implementation is, and not whether the implementation is correct or not. Programs that won't be executed correctly without the priority specification are incorrect KL1 programs.

If priority specification should be strictly obeyed in a parallel implementation, the whole system must be inspected in each execution step to ensure that there exist no goals with higher priority. If this should have been done, the locality of computation would be totally lost.

2.4.2 Priority Specification

Two levels of priority specification are provided in KL1: Shōen by shōen coarse specification and goal by goal

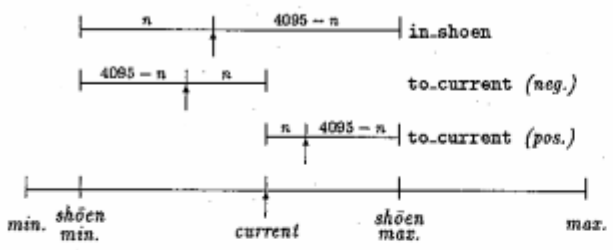


Figure 2: Priority Pragma

finer specification.

When a meta-level program controls the priority of object-level programs (for example, when the operating system specifies the priority of user programs), shōen by shōen specification is the recommended way. Priority of the shōen is specified as the minimum and the maximum priority allowed for goals and children shōens in it. They are given as additional arguments of the *execute* primitive when the shōen is created[11].

When the object-level program controls the execution of itself depending on detailed object-level knowledge (for example, when heuristics are used), goal by goal specification may be appropriate. The priority of a goal can be specified by affixing a priority pragma to the invocation of the goal (see below for details). When no priority pragma is given, the priority of the parent goal is used as the default value. When a new shōen is created, its top level goal will have the maximum priority allowed in the shōen.

In both types of specification, the priority is specified relative to the priority minimum and maximum of the immediately surrounding shōen and the *current* priority (the priority of the goal currently being reduced). There are two ways in this relative specification.

- Specifying by ratio in the range of priority minimum and maximum of the surrounding shōen.
- Specifying by ratio in the range of the *current* priority and the maximum (or the minimum, when a negative value is specified) priority of the surrounding shōen.

The ratios are given by an integer n, where $n/4095$ is the actual value of the ratio[12].

For example, priority pragma for goals is given as follows.

[11] In the current implementation, once a shōen is created, its priority range cannot be changed afterwards. An alternative implementation which allows it is being investigated.

[12] It is better understood as representing a fixed-point real number.

```
p(X,W) :-
  q(X,Y)@priority_in_shoen(4095),
  r(Y,Z)@priority_to_current(-16),
  s(Z,W).
```

Here, the goal `q(X,Y)` will have the highest priority allowed in the shōen, and the goal `r(Y,Z)` will have the priority somewhat lower than the current priority, i.e., that of the parent goal `p(X,W)`. As `s(Z,W)` has no priority pragma, it will have the same priority as the parent goal.

Specifying priorities relatively, rather than absolutely, has the following merits.

- Local relative priority specifications are properly respected without any change when the program is run in a shōen with more global priority specification.
- Different implementations may have different physical priority ranges. Relative priority specification can be free from such implementation dependency.

2.4.3 Implicit Fairness?

In an implementation of a parallel language with limited computational resources, execution of object-level programs may make the meta-level program wait for at least a while. In a naive implementation, even if the meta-level program is about to stop the object-level program, the meta-level may not be executed forever, waiting for an infinite loop in the object-level program to terminate and yield the required resources back.

A simple method to solve this problem is to introduce implicit *fairness* to the scheduling strategy. A *fair* scheduling here means, any goals that are ready to be executed will be reduced at least by one step sometime in a finite time period. To implement this, breadth-first scheduling or introducing depth limit to depth-first scheduling has been proposed.

On the other hand, the problem can also be solved by the priority mechanism, by specifying the priority of object-level programs explicitly not to become higher than the meta-level program. Implicit fairness is not required in this case. An important merit of not adopting implicit fairness is that, when all the ready goals have the same priority (that means almost all the time when a simple application program is running without much communicating with the operating system), the implementation is allowed to choose the most efficient scheduling, fully utilizing the locality between goals. Fortunately, simple depth-first scheduling is known to yield the best results usually.

To assure that the meta-level program can stop infinite loops of the object-level programs, the priority must be respected at least a little by the implementation. What is requested is that a goal with the highest

priority should be reduced at least by one step within a finite time period.

2.4.4 Priority Between Clauses

In the original GHC, when multiple clauses can be used for reduction of a goal, the implementation is allowed to choose whichever clauses for the reduction. This semantics allows the implementation to choose the most efficient reduction strategy. Consider, for example, the following two clauses.

```
m([W|X], Y, WZ) :-
    WZ = [W|Z], m(X, Y, Z).
m(X, [W|Y], WZ) :-
    WZ = [W|Z], m(X, Y, Z).
```

When the first argument is on a remote processor and the second argument is already known to be instantiated to a list cell on the currently executing processor, the implementation is allowed to choose the second clause without even trying to access the first argument.

With this mechanism, however, preference between clauses cannot be specified in the program. For example, again in an alpha-beta search program, when a new maximum (or minimum) value is found by a child node, that data should be used to control other children nodes more efficiently. If that value is not yet available, the goal will continue with the already known maximum (or minimum). In this case, the clause waiting for the report of new maximum/minimum values should have priority to other clauses.

Specification of priority between multiple candidate clauses are thus introduced to the KL1 language. For example, the above merger program will become a priority merger by adding priority specification between clauses as follows.

```
  m([W|X], Y, WZ) :-
      WZ = [W|Z], m(X, Y, Z).
alternatively.
  m(X, [W|Y], WZ) :-
      WZ = [W|Z], m(X, Y, Z).
```

Note that the second clause is tried not only when the first clause fails but also when it *suspends*. Thus, it is *not* something like the sequential OR feature seen in Parlog or the *otherwise* feature in Concurrent Prolog.[13]

Similar to priority between goals, the implementation does not have to fully obey the priority specification. In the above example of the priority merger, the second clause *may* be chosen even after the first argument is already instantiated to a list cell. Execution efficiency will be considerably lost in implementations on parallel hardware if the specification must be obeyed perfectly. What is requested is that the first clause will be chosen sometime within a finite time period even when the second clause can be chosen for infinitely many times.

In the implementation level, the clause with priority will always be used for the reduction as far as all the required data are available within the processor reducing the goal. Otherwise, the second clause may be used but fetching of the data required for reduction by the first clause from remote processors is initiated, even though it is not required for this particular reduction. It will eventually make the first clause ready to be chosen.

2.5 Quantitative Control

As described above, information such as the termination of the computation or emergence of exceptions are notified to the meta-level through the report stream of the shōen. Execution control of shōens to suspend, resume or abort the execution of a shōen is also provided through the control stream.

These features are *qualitative* in that the controlled shōen either can proceed or not. To control the meta-level behavior of the object-level program more into its detail, features to *quantitatively* control the object-level computation are required, in addition to the qualitative control features. Quantitative control is to control meta-level quantities of computation, such as *how much* computation should be allowed for a shōen.

The quantitative control features cannot be efficiently realized easily without certain language level support. The KL1 language thus provides not only the qualitative meta-control, but also quantitative meta-control features as its language primitives.

2.5.1 Principles

The shōen, which is the unit for qualitative execution control, is also used as the unit for quantitative control. Using the same shōen mechanism is profitable in making additional overhead smaller.

Although each program runs differently depending on the problem and the algorithm, lower level notions such as how much processing time or how much memory area the computation consumes can be reasonable common measures for all kinds of programs. Thus, such lower level quantities are controlled by the feature.

The simplest method may be to report each execution step in the object-level to the meta-level. This method, however, requires large a amount of communication between two levels. In KL1, the meta-level sets some limit to the resource consumption of the object-level program, and the object-level only reports the *resource_low* status

[13] The KL1 language also provides the sequential OR feature. The sequential OR and *otherwise* directives can, however, be replaced by writing negation of the guard parts of all the preceding clauses. Thus, although such language constructs are quite useful, they merely provide a syntactic convention without extending the essential descriptive power of the language.

to the meta-level when the allowed amount of resource has *almost* been consumed up. If the resource consumption actually reaches the limit, then the execution is suspended until the limit is raised by the meta-level program.

Receiving the *resource_low* report, the meta-level program can choose from the following.

- Add some more to the limit to make the object-level computation continue.
- Abort the object-level computation by sending an *abort* message to the control stream.
- Suspend that computation but freeze the computation status as it is (by simply *not* adding any more to the limit), until possible future resumption.

When the computation should be continued, the meta-level program will add some more amount to the resource consumption limit. However, there may be some delay due to computation required for such a decision or for communication between processors. Thus, the report is made somewhat before the given resource is completely exhausted. This allows pipelined resource supply.

Setting small resource consumption limit and adding to it a small amount frequently, more accurate resource consumption control is possible. However, it may require more resource handling overhead. The accuracy of resource management is a parameter of the system that can be defined by the meta-level program considering this trade-off.

2.5.2 Nested Resource Management

The resource management principle of KL1 is also based on the assumption that shōen is a machine-level interpreter. The virtual interpreter is assumed to be counting the resource consumption.

The virtual interpreter is considered to be made so efficient that interpretation of a program consumes only the same amount of the resource as when the interpreted program is executed directly.

When shōens are nested, the inner interpreter is interpreted by the outer interpreter. If the outer interpreter detects that the resource consumption limit has been reached, it will suspend interpretation until some more resource consumption is allowed. Naturally, the inner interpreter is forced to stop there.

With this semantics, the meta-level program can control resource consumption of the object-level programs without being aware of lower level meta/object layers.

2.5.3 Controlled Quantities

The following are the candidates of resources controlled by the mechanism.

Time: How much CPU time can be used for the execution of the shōen.

Space: How much memory can be used for the execution of the shōen.

In stead of measuring the CPU time, the current implementation counts the number of reductions as its estimate. The current implementation does not count the memory consumption.

For about memory consumption, the garbage collection mechanism makes fair management difficult. As the same memory area can be made available again by garbage collections, such reuse should not be counted as *consumption*. However, a data structure created in one shōen may be passed to another and then garbage-collected. Memorizing all such data transfer has too much overhead. In addition, notifying garbage collector of the affinity of memory blocks to shōens will be quite costly.

Another difficulty is in unbalanced resource consumption. When multiple processors are available, one program may consume much memory on one processor but not on others. To cope with such cases, memory consumption management in terms of total amount of allocated memory is never enough; some amount of memory consumption in one processor is not equivalent to that amount of consumption in another processor. This problem is left over as a further research theme.

2.5.4 Resource Management Messages

Communication required for resource management is effected through the control and report streams of shōen.

When the resource consumption limit is reaching in a shōen, a *resource_low* message is sent to the report stream. Adding some amount to the resource limit of a shoen is effected by sending an *add_resource(Amount)* message. In response to this message, a *resource_added* message is sent to the report stream. Watching the order of this message and a *resource_low* message, it is possible to know whether resource left is found to be low *before* the resource addition is made or it became low even *after* the addition.

To query the resource consumption status, the control message *statistics* can be used. As a direct response to this message, the *statistics_started* message is sent back from the report stream. Sometime after that, a *statistics(Status)* is sent to the report stream, bringing the resource consumption statistics in its argument. The reported resource consumption status is that of sometime in between the two time points at which these messages are issued.

2.6 Executable Code

To be a self-contained computer system, programs in the system must be handled by the system itself. The KL1 language thus provides features to handle executable codes as data.

2.6.1 Module and Code Data Types

A block of object code for KL1 programs can be handled as a data object of type *module*. One module may contain executable codes for several predicates. Predicates declared to be *public* in the source program are registered in a certain table in the corresponding module data object, which can be accessed using a built-in predicate. Such predicates entries can be handled as a data object of type *code*. Other predicates are local and can be invoked only from inside of the module.

Module and code data objects are treated basically the same as data objects of other types; they can be passed as arguments, stored in structures, garbage-collected if no access path remains to them, etc.

A module data structure consists of two parts: *GC part* where pointers to other data are stored and *non-GC part* where only atomic data, mainly executable KL1-B code (Kimura and Chikayama 1987), are stored. All accesses to data outside of a module is made via pointers stored in the GC part. The GC part can contain pointers to uninstantiated variables. Thus, when created, modules can contain an invocation of a module that is not defined yet. Accesses to such a module before its definition will be simply suspended. Later instantiation of the variable will make it proceed. It is along the principle that executable codes should be treated the same as other data objects.

The non-GC part of modules contains lower-level machine code. Thus, if module data objects can be arbitrarily created, any protection mechanism above the KL1 language level may be violated. To avoid this, the built-in predicate for creation of modules is not made available to user programs. This is realized simply by not including it in the built-in predicate table of the compiler when it is in the application program compilation mode. User programs can only create module data objects by asking the PIMOS for compilation of source programs, which will never generate problematic codes.

Creation of a module is suspended until all of the elements of its non-GC part becomes instantiated, and they are fully dereferenced during the creation; the low-level execution mechanism can safely assume that executable machine code is already there.

2.6.2 Higher Order Mechanism

The code object can be used for execution by the built-in predicate *apply*. The *apply* predicate takes two arguments: The code for a predicate and a vector of arguments. When either one of these are still uninstantiated on an invocation of *apply*, it is suspended until both get instantiated.

The *apply* built-in predicate is a higher order extension to the language. The shōen feature described above also takes this higher order approach for treating executable code. Some other concurrent logic programming languages adopt meta-level mechanism, in which usual data structures such as `p(X)` are treated as executable code (Clark and Gregory 1986). This approach, however, assumes that the mapping from predicate name (the atom p) to the corresponding executable code is available completely in the language implementation level.

When the higher order invocation mechanism is available, the meta-level feature can be implemented in the software, allowing full flexibility in name/code mapping to the software. Section 4 describes how such mapping is implemented in the PIMOS.

2.7 Other Extensions

In addition to the extensions required for meta-level programming, several other extensions to the original GHC language are made in the KL1 language. They are mainly for providing efficient primitives, which, however, affected the design of the PIMOS considerably.

2.7.1 Random Access Structures

Many of the implementation level optimization of the KL1 language are based on the low-level mechanism that distinguishes multiple and single reference paths to data objects. Such information is kept in pointers using one bit tag called the multiple reference bit (MRB) (Chikayama and Kimura 1988). At the implementation level, the information is used mainly for incremental garbage collection. When the sole reference path to an object is known to be required no longer, that object can be reclaimed.

This single reference information can be utilized to implement efficient random access structures. The KL1 language has one-dimensional array structure data type called *vector*.[14] Updating an element of a vector can be effected by the following built-in predicate.

set_vector_element(OldV, N, OldE, NewE, NewV)

The arguments have the following meaning.

OldV: The original vector structure.

N: Index of the element to be updated.

OldE: The *N*th element of the original vector.

[14] Usual functor structures such as "`f(X)`" are also represented using the vector structures such as "`{f, X}`".

NewE: The Nth element of the newly created vector.

NewV: The newly created vector with the same length and elements as the original vector, except that the Nth element is replaced with *NewE*.

As far as the semantics is concerned, this predicate has no side-effects. It allocates a copy of the original vector with Nth element altered to *NewE*. A naive faithful implementation of this, however, requires time and space proportional to the size of the vector. When the reference path to the original vector is known to be the last one, that data structure can be destructively updated in constant time with no memory allocation, without disturbing the pure semantics.

The semantics of the language is not affected by the existence of such an optimization. However, when such an operation is known to be efficient, use of random access structures is strongly encouraged and the programming style of KL1 may become drastically different.

There exist sequentiality between element accesses to (physically) the same (but logically different) vector. The updated vector can be accessed only after the update procedure is completed. The execution of the *set_vector_element* predicate is automatically suspended until its first and second arguments become instantiated. Note, however, that the *NewE* argument need not be instantiated on update. The corresponding element of the new vector will simply become that uninstantiated variable. Consider, for example, the following process-like program.

```
table([update(N,X)|S], OV) :-
  set_vector_element(OV, N, OE, NE, NV),
  compute(OE, X, NE),
  table(S, NV).
```

The `compute` predicate computes the new element value from its previous value and the data supplied with the message. In a parallel implementation, the call of `compute` and the recursive call of `table` can be executed in parallel. If the next message is updating the same element, it will be suspended naturally because data required for `compute` is not available yet. If it is updating a different element, however, `compute` for that message can be executed in parallel with the first one.

2.7.2 Merger

As stream-like communication using list structures is a frequently used programming technique (Shapiro and Takeuchi 1983), the efficiency of stream merge operation can be a key of the overall efficiency of the system. Thus, a stream merging mechanism is built into the system. Again, it does not affect the semantics of the language but does affect the programming style considerably.

A merger process is created by the following built-in predicate.

merge(In, Out)

Immediately after the invocation, the merger is merging only one input stream to the output stream (this, of course, is not a merger yet). Its semantics can given by the following clauses.

```
merge([],    O ) :- O = [].
merge([X|I], XO) :- XO = [X|O], merge(I, O).
```

More input streams can be added by unifying the input stream argument to a vector structure whose elements are streams to be merged in. This feature can be described by the following additional (infinite number of) clauses (curly brackets are used to denote *vector* structures).

```
merge({},          O) :- O = [].
merge({I},         O) :- merge(I, O).
merge({I1,I2},     O) :- merge(I1, I2, O).
merge({I1,I2,I3}, O) :- merge(I1, I2, I3, O).
...
```

The `merge` predicates with three or more arguments also have clauses for increasing the number of merged streams, in addition to the clauses for actual merging.

This description in the KL1 language gives the semantics but the actual implementation is of course quite different. It is highly optimized using the MRB information.

For increasing the number of input streams, the vector structure is used directly as the argument of merge.[15] This scheme is advantageous to the scheme using a special message for that purpose (Ueda and Chikayama 1984) in that no reserved *message* is used; addition of merged streams is specified by the argument itself rather than the *car* part of the list structure, where a message should be. Uninstantiated messages can go through the merger, because, although being uninstantiated, they are known to be a message that goes through the merger and not something controlling the merger. The basic mechanism of the stream communication using list structures is that, communication control is done by the *cdr* part which is a list cell or nil, and the *car* part brings a message. The scheme adopted in KL1 keeps this principle of using only the *cdr* part for controlling the communication.

3 INPUT AND OUTPUT

This section describes how physical I/O devices are modeled in KL1, what kind of logical I/O interface the PIM-OS provides to the user, and how they are realized.

[15] Any structures other than lists could have been used here. The vector structure is used only because it is the most efficiently handled structure.

3.1 Model of Physical Devices

I/O devices can be modeled by processes of KL1. In a certain layer of the PIMOS, I/O devices behaves the same as KL1 processes.

A one-line display device can be modeled by the following KL1 clauses.

```
line_display([display(S)|R], E, _) :-
  wait(S) |
  line_display(R, E, S).
alternatively.
  line_display(R, E, S) :-
    E = [photons(S)|E1],
    line_display(R, E1, S).
```

The device is always radiating photons messages describing the string displayed to the *ether* (via the stream E in the program). The displayed string can be changed by receiving a request display from the host (via the stream R) with its argument being a new string.

A character input device can be modeled by the following KL1 clause.

```
char_input(O, [stroke(C)|K]) :-
  O = [C|O1],
  char_input(O1, K).
```

Unlike in the case of output devices, the communication stream (O in this case) flows from the device to the host. The device simply sends all the characters typed in to its output stream. Buffering is implicitly effected by the output stream.

An explicitly buffered character input device can be modeled by the following KL1 clauses.

```
char_input(R, [stroke(C)|K], B, T) :-
  T = [C|T1],
  char_input(R, K, B, T1).
char_input([get(X)|R], K, [C|B], T) :-
  X = C,
  char_input(R, K, B, T).
```

In this model, direction of the stream is from the host to the device. Characters typed in are buffered in this process, using the third and the fourth arguments as a difference list representing the buffer. They are sent to the host on get request by unifying the argument of the partially instantiated request message.

Among the above two models for input devices, the PIMOS employs the latter, mainly because devices for input, output and both can all be handled uniformly. The request stream from the host to the device is called the *device control stream*.

3.2 Device Control Scheme

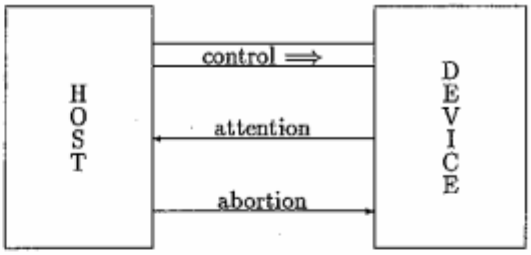


Figure 3: Communication Paths to a Physical Device

3.2.1 Completion Status

All the I/O command messages sent to a device control stream have an argument to which the device process unifies a value indicating whether the command was normally completed or not. This argument is called the *completion status* of the command. Synchronization with the completion of a command can also be possible by waiting instantiation of this argument.

3.2.2 Interrupt

With only the control stream, there is no way to send information actively from the device process to the host. Besides the control stream, a reverse direction communication path called the *attention line* is provided. The attention line is actually a shared variable among the host process and the device. The device may instantiate it when asynchronous communication to the host is required.

3.2.3 Command Abortion

Sometimes, cancellation of commands already sent to the control stream is desired. In the PIMOS, all the I/O devices have an additional communication path from the host to the device, called the *abortion line*, which is actually a shared variable among the host process and the device. When the host instantiates this variable, the device aborts the execution of the command under execution and skips any subsequent commands in the device control stream until a *reset* message appears in the stream. The completion status of the aborted and skipped commands will become *aborted*.

A reset message has the following arguments.

Abort: New abort line.

Attention: New attention line.

Status: Completion status of the *reset* command.

After the reset message is received, subsequent commands are processed normally using the new abort and attention lines specified in the message.

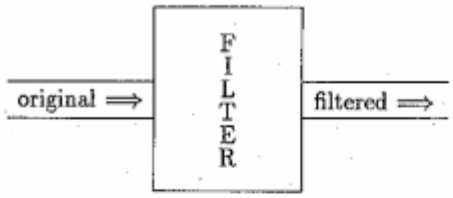


Figure 4: Filter

3.3 Lower Level Implementation

In the current implementation on the Multi-PSI version 2, the above-described communication protocol (based upon the model of physical devices as KL1 processes) is almost faithfully realized by the I/O front-end processors. The KL1 language implementation is not aware of the existence of such specialized front-end processors; the front-end and the host communicate to each other using the same protocols used in the communication between two processors of the host. In this scheme, the message handling mechanism for KL1 language implementation must be implemented also on the front-end processor. The merit of the scheme is that it makes the language implementation simpler, which was quite important in this prototype implementation.

In other implementations in the future, realizing lower levels on the host machine and making I/O processor simpler may become advantageous. In that case, a lower level model of devices with clean semantics will be required.

3.4 Logical Devices

Logical devices are provided by the processes called *device drivers* of the PIMOS. A higher level abortion mechanism which allows retrying of once aborted commands is provided here.

The device driver is a kind of *filter*. Filters are processes that receive messages from its input stream, process them somehow depending on the nature of the filter, and send the processed message to the output stream.

The device driver remembers aborted I/O commands. They can be sent again to the physical device by a *resend* command. Alternatively, that memory can be cleared by a *cancel* command. This decision can be delayed arbitrarily long, even until after another abortion. Thus, multiple such memories for multiple groups of aborted commands are required. The *reset* message in this level has an additional argument *ID* to which the identifier of the immediately preceding aborted command group is returned. This identifier is used in *resend* or *cancel* commands to identify an aborted command group.

This feature is quite useful in programs where two or more tasks share one device. For example, programs running under a command interpreter shell often share a display window with the shell itself for standard input/output. When such a program is suspended by an interruption, there may be multiple I/O requests already sent to the window logical device but not processed yet. In such a case, the shell *aborts* the processing (through the abortion line) and sends a *reset* messages to the window device driver, and then uses the window normally for its own purpose. All the I/O requests of the suspended program are also suspended and remembered in the device driver process, rather than being discarded. When the suspended program is to proceed again, sending a *resend* message to the device driver will continue the processing of the suspended I/O requests. When it is to be killed, a *cancel* message is sent instead.

3.5 Buffering

The I/O commands can be designed to send one command for each character or similar small units, which may be convenient for most application programs. Applying that fine-grained protocol to all the communication channels in the system, however, the communication overhead may become problematic. Where there is a considerable per-message overhead for communication, for example, in communication between the host machine and the I/O devices, one message should bring as much data as possible, as far as communication delay will not become a problem, to attain higher throughput. To realize this without changing the end-user interface, the well-known technique of *buffering* is widely used in conventional operating systems. Simply by buffering n characters, the per-message communication overhead can be reduced to $1/n$.

The PIMOS provides the buffering mechanism in the process called *I/O utility filter*, which is a filter placed between the device driver and the user program.[16] Thus, the command protocol of this buffer is the only one that casual users are concerned. Buffering can be made quite efficient using updatable random access structures (in this case, updatable character strings).

The size of the buffer is a parameter of the system which should be determined depending on the hardware parameters. In addition to the buffering feature, this filter also provides parsing and unparsing features for operator precedence grammars.

4 MANAGEMENT OF PROGRAMS

In the KL1 language level, atoms do not have any association with their name strings nor any other properties such as executable codes. They are merely *identifiers* in its original meaning. It is the PIMOS that associates atoms and their name strings or any other correspond-

[16] There are other filters in between the device driver and the I/O utility filter, which will be explained below.

ing properties. This section describes the databases the PIMOS provides for storing such information and how they are used.

4.1 Atom Name Database

Atoms and their names (character strings) are associated by the *atom name database* provided by the PIMOS. This database is accessed when such an association is required; for example, on Prolog-like read or write operations. The database is implemented as a KL1 process, which maintains two hash tables: One for mapping names to atoms and the other for the reverse. These two tables are always kept consistent. Hash tables can be quite efficiently implemented using the randomly accessible and updatable array structures described in 2.7.1.

If this single database is used by all the programs running under the PIMOS each time they need the information, the process realizing the database can be a performance bottleneck of the system. Fortunately, the atom name database is monotonic; new atom/name pairs may be added to the database but there is no deletion nor update. Thus caching of the atom name database is quite easy.

A cache database is created as an empty database. When a query is made to a cache, the query is sent further to the central atom name database (or another cache database) and the answer is remembered in the cache. Any subsequent queries made on the same atom can be answered without accessing the central database. No other synchronization is required.

Note that the association of atoms and their names are provided solely by the software, rather than the language implementation. Thus, the atom/name association provided by the PIMOS is merely the *standard* and not the only possible one. Users can use their own database for specific applications. It is also possible to create unique atoms not associated with any names, if only *identity* is of interest.

4.2 Module Database

As described in 2.6.1, a block of object code for KL1 programs is handled as a data object of type *module*. The *module database* provided by the PIMOS associates module names (atoms rather than name strings, actually) with the corresponding module data objects.

Caching the module database is not as easy as that of the atom name database, because modules may be updated. Non-deterministic parallelism makes keeping of consistency difficult. Fortunately, access frequency is considered to be much lower in this case. Thus, the PIMOS currently does not provide any caching mechanism for the module database.

A module which is not defined yet can also be registered to the module database. Such a module is represented by an uninstantiated variable. When a query to obtain such a module is made, that variable is returned. Accesses to such a module will be simply suspended, as described in 2.6.1.

Note that the association of modules and names provided by the module database of the PIMOS is merely a *standard* also and not the only one, as in the case of atom/name association. When, for example, a higher level language system is to be built upon the PIMOS, it may or may not use the standard association.

When the number of users concurrently using the PIMOS increases, the module database may become a performance bottleneck. In such a case, providing a private module database for each user may be advantageous. Commonly used modules, such as those provided by the PIMOS, should be stored in a common database which does not allow any update, to enable caching in personal module databases. The look-up mechanism will be similar to the *package* system provided by Common Lisp (Steele 1984).

4.3 Linking Modules

There are two types of linkage between modules.

The basic linkage mechanism is *fixed* linkage provided by the language system. When linkage is made this way, a module containing a invocation of a predicate in another module has a pointer to the invoked module object.[17] This is the most efficient mechanism provided for invocation of a predicate in a foreign module.

A drawback of this efficiency is that modules cannot be updated independently. For example, when a module A calls another module B, updating the module B to B' will not make the module A call the new module B'. The newly created module B' is merely replacing B in the module database, without changing already existing pointers to the module B elsewhere. In such cases, the module A must be linked again with B'.

In the more flexible linkage mechanism implemented in KL1 software, modules are not directly referenced but designated by their *names*. Each time an invocation is made using this linkage scheme, a query to the module database is made. When this *soft* linkage is used, each module can be updated independently. Efficiency drawback, of course, is not small.

Fixed linkage is normally used. Name linkage is used in cases where programs are in anyway invoked by their names. For example, the command interpreter shell uses this on invoking programs that run under it. Name linkage may be profitable also in the program development phase, where linkage efficiency may be more important than execution efficiency.

[17] There can be many copies of one module on different processors, but they all are logically equivalent.

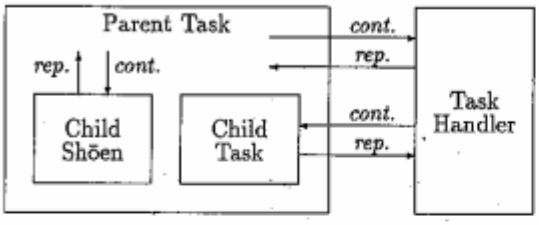


Figure 5: Task and Shōen

5 RESOURCE TREE

Controlling computational resource is the most important role of an operating system. Using the shōen feature, consumption of basic resources such as execution time can be controlled. There are, however, other resources, such as I/O devices, which are not controlled by the language primitives and should be controlled by the operating system.

This section describes how management of such resources is realized in the PIMOS. The most crucial part of the resource management is in releasing resources allocated during some computation on abortion of that computation.

5.1 Resources

The shōen feature provided by the KL1 language is capable of controlling basic resources such as execution time. The kinds of resources controlled by the shōen feature is called *language-defined resources*. On the other hand, other resources such as I/O devices cannot be controlled only by the shōen feature. Such resources are called *OS-defined resources*.

5.2 Tasks

Tasks are units of resource management in the PIMOS. Tasks are a shōen specially recognized as a task by the PIMOS.

As the shōen feature is provided by the language, shōens can be arbitrarily created at any time by any program. Tasks, on the other hand, can only be created by asking the PIMOS, because it is the only way a shōen can be recognized as a task by the PIMOS.

The control and report streams of a usual shōen are directly connected to its creator, while those of a task are connected to a PIMOS process corresponding to the task, called *task handler*. The creator of the task can only indirectly control the task and receive reports of the task through streams connected to the task handler process.

5.3 Resource Loop

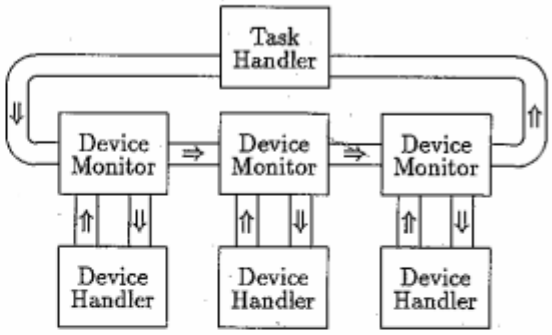


Figure 6: Resource Loop

When a task is aborted, all the OS-defined resources allocated to the task are freed. This is essential to allow abortion of tasks safely without disturbing subsequent processing.

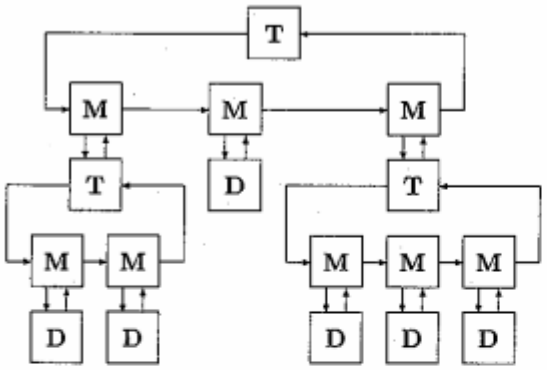
To realize this, all the OS-defined resources allocated to the task must be remembered somehow. All accesses to OS-defined resources from user programs are made through a stream that is connected to a process in the PIMOS, called *device handler*. The device handler process is a filter, through which various requests are sent to the device driver. This handler processes can thus control users' accesses to the OS-resources.

All the OS-defined resources are associated with another PIMOS process called *device monitor*. The monitor processes for resources allocated in a task are connected by a stream in a loop structure called *resource loop*.[18] The monitor process and the handler process have communication streams in both ways. Resources allocated in a task can be released on its termination by sending a message notifying the termination via the resource loop to the monitors, and then to each corresponding handlers. Handlers will close their output streams when when the message is received. When a resource is released individually (when a file is closed, for example), that resource can be eliminated from the loop using the well-known short-circuit technique (Hircsh *et al.* 1987). Queries on the resource allocation status can also be processed along the same path.

A monitor and the corresponding handler are made as distinct processes in the current implementation for keeping the modularity of the system; all the monitors are identical but the handlers depend on the device they handle. It may be possible, however, to merge these two processes.

5.4 Resource Tree

[18] A loop structure similar to the resource loop of PIMOS can also be found in the Logix operating system (Hircsh *et al.* 1987) for controlling user programs. In case of Logix, however, the unit of control is each goal to be reduced, as there is no notion of goal groups such as shōen in the language level.



T: Task Handler; **M:** Resource Monitor; **D:** Device Handler

Figure 7: Resource Tree

In the PIMOS, tasks are also considered to be OS-defined resources. A task handler is a kind of device handler whose corresponding device happens to be a shōen. Tasks are different from other devices in that they may have children resources.

As children resources of a task can be tasks again, all the device monitors and handlers form a tree of resources. This is called the *resource tree*. All the resource management of the PIMOS is through this resource tree.

6 COMMUNICATION MECHANISM

As there is only one variable binding environment, the operating system PIMOS and user programs can share variables. Communication between the PIMOS and user programs is made through such shared variables. The user program instantiates a shared variable to a data structure for making a request to the PIMOS, and the PIMOS instantiates certain elements of the structure to return values to the user program.

This section describes how such communication is made, how the communication path is initially established, and how the communication is made in a fail-safe way to protect the PIMOS from accidental or intentional errors of application programs.

6.1 Basic Communication Mechanism

The communication mechanism between the PIMOS and the user programs is based on the scheme proposed in (Shapiro and Takeuchi 1983).

The structure of the top level of the PIMOS may be as follows.

```
boot :-
  pimos(S),
  execute(user(S), ...).
```

The variable S is shared between the user program and the PIMOS and used as a *stream* for communication between the user program and the PIMOS.

For simplicity, we assume here that the PIMOS provides only one device: A character input device. In this case, the clause in a predicate of PIMOS for handling one character input requests may be defined as follows.

```
pimos([get(C)|S]) :-
  read_keyboard(C1),
  C = C1,
  pimos(S).
```

Here, the `read_keyboard` predicate is assumed to do the physical input procedure and unifies the typed-in character with its argument. When a message `get` is received, physical input operation is performed and the typed-in character is unified with the argument of the `get` message.

In the above program, the user program may be as follows.

```
user(S) :-
  S = [get(C)|S1],
  user1(C),
  ...
user1(0'a) :- ...
user1(0'b) :- ...
  ...
```

The user program sends a partially defined `get` request message to the operating system, and then determines its further processing depending upon the result. The predicate `user1` implicitly waits for instantiation of the variable to which the operating system is returning a value by unification.

The order of the commands sent to one message stream is kept in the stream, for it is determined by the data structure, not by the order of operation.[19] This guarantees, for example, two messages to be displayed are displayed in the desired order.

Accesses to databases provided by the PIMOS are also made in the same manner. In this case, the stream obtained by request the PIMOS is an access path to a database, in which commands are ordered.[20]

6.2 Establishing Communication Paths

For communication using the above-described method, the part of the user program which requires some service of the PIMOS must have a stream connected to the PIMOS (or one that merges into it). As there is no notion of

[19] When mergers are inserted, the order of messages from originally different streams becomes non-deterministic.

[20] When two or more access paths are created, synchronization of two access paths should be made by the completion status argument of command messages.

global variable in the language, such a stream must be passed all through the chain of invocations from the top level to where actual communication is required. This overhead may be too large if communication is needed only in rare exceptional cases; only in case of error reports, for example.

The PIMOS provides an alternative way of establishing a communication path to the PIMOS. This can be done by deliberately generating an exception using the *raise* primitive (see section 2.3.4). By specifying an appropriate *tag* in this this *raise*, the exception report indicating the request goes directly to the report stream of the shōen used for realizing a *task*. The report stream of a task shōen is monitored by the task handler process of the PIMOS, and there, the request is processed. Thus, shōens in user programs can nest arbitrary number of levels without losing the direct availability of the services of the PIMOS.

It is also possible for a user program to create a shōen specifying a tag corresponding to some requests to the PIMOS. This way, all or part of the requests to the PIMOS can be caught by the program monitoring the report stream of such a shōen. If the monitor program emulates the PIMOS, the program in the shōen runs exactly the same as when it is directly run under the PIMOS. This is the way virtual machine operating systems are implemented under the PIMOS.

There are three layers of communication streams between user programs and the PIMOS.

Device Level: This is the lowest level where concrete I/O command messages are sent.[21]

Device Request Level: This is the level where command messages to obtain device level streams are sent. For example, a file request stream accepts messages asking to open files and return the device level stream connected to the file.

General Request Level: This is the top level where command messages to obtain device request streams are sent.

What is directly obtained by *raising* an exception is a general request stream. All the services provided by the PIMOS are available through this general request stream.

6.3 Protection Problems

With the simple mechanism described above, however, intentional or accidental error in user programs may cause a system failure. This section describes the problems in shared variable communication between user programs and the operating system.

[21]Some devices may also accept commands which creates a new device. For example, a file directory device can create a file device.

6.3.1 Multiple Writer Problem

As far as the user program is properly written as described above, there will be no problems. If, however, an erroneous user program such as follows is executed, a system failure may take place.

```
user(S) :-
  S = [get(C)|S1],
  user1(C),
  ...
user1(C) :-
  C = 0'a,
  ...
```

Here, the user program unifies the variable C. If this happens *before* the unification "C = C1" in the PIMOS, the unification in the PIMOS may fail. Even if it were possible to check that the variable C is uninstantiated *immediately* preceding the unification[22], the unification in the user program can be executed in between the check and the unification in a parallel system.

The Parlog Programming System (PPS) provides a simple solution to this problem (Foster 1987). The problem arises because the value-returning unification is executed in the PIMOS. The PPS solution is to make the unification done in a metacall (a mechanism similar to shōen). Using this scheme, the code for the PIMOS will be as shown below.

```
pimos([get(C)|S]) :-
  read_keyboard(C1),
  wait_and_unify(C1, C),
  pimos(S).
wait_and_unify(C1, C) :- wait(C1) |
  execute(C = C1, ...).
```

Waiting the value of C1 is essential because otherwise the order of the invocation of the predicate read_keyboard and the execution of the shōen will not be defined in the language, and thus, without waiting for the value of C1, the unification in the shōen can be executed *before* the invocation of the read_keyboard predicate. This solution is simple but requires frequent metacall invocation; one metacall per one communication from the operating system to the user is required. The metacall mechanism cannot be optimized easily by compilation and other optimization efforts. Thus, this solution may be reasonable for interpretive implementations of the language where the metacall mechanism is relatively inexpensive, but may not be the best when the reduction mechanism is highly optimized.

The problem may also be solved by introducing the *atomic commitment* mechanism provided by Concurrent

[22]The KL1 language does not provide a built-in predicate for variable check, such as *var* in Prolog. Such a predicate can only guarantee that its argument was not instantiated sometime before in parallel implementations.

Prolog (Shapiro 1983). Implementation with atomic commitment mechanism, however, may be not as efficient as one without it. The problem is that the cost of atomic commitment mechanism is not only in the communication with the operating system but also in every goal reduction in the system where such mechanism is not required.

6.3.2 Forsaken Reader Problem

Another problem appears when the user program fails to instantiate a shared variable inspected by the PIMOS. For example, consider the following clause.

```
user(S) :- S = [_|S1], ...
```

As the message to the PIMOS is not instantiated, the PIMOS process will wait for it to be instantiated forever. When the message has an argument which specifies more details of the request (a character code to be output, for example), the same problem may occur in the argument level, too. A similar situation also arises when the execution of the user program is aborted using the shōen mechanism described above, even if the user program is properly written.

This problem cannot be solved using the shōen mechanism nor the atomic commitment mechanism.

6.4 Protection Filter

To solve the above-described problems, a filtering process called the *protection filter* is inserted in the stream between user programs and the PIMOS. This filter is executed in the user shōen rather than in the PIMOS.

The user sends messages to the protection filter stream, not directly to the PIMOS. The protection filter translates the user messages into a different form which does not cause failure in the PIMOS, and sends it to the PIMOS.

The concrete functions of the protection filter are as follows.

- It waits for instantiation of variables which the user should instantiate. The message is sent to the PIMOS only after that. Thus, when a message is sent to the PIMOS, the values of these variables are guaranteed be instantiated. For structures, it waits for instantiation of only certain elements of it which are required to be instantiated, allowing partially defined messages to pass through the filter.
- It replaces variables which the PIMOS should instantiate with new unbound variables. It also initiates processes each of which waits for instantiation of one of the newly created variables, and then unifies the corresponding original variable with it. Thus, all value-returning unifications in the PIMOS will always be with unbound variables, which will never fail.

The protection filter for the above example will be as follows.

```
filter([get(C)|S], OS) :-
  OS = [get(C1)|OS1],
  wait_and_unify(C1, C),
  filter(S, OS1).
wait_and_unify(OSV, UserV) :-
  wait(OSV) |
  UserV = OSV.
```

The filter process will not proceed until the message becomes instantiated to the form `get(C1)`; the `wait_and_unify` predicate unifies the variable supplied by the user program (`UserV`) with the variable to which the operating system returns the result (`OSV`) only after its instantiation.

The key point here is that the protection filter process is in the shōen of the user and thus the unification "`UserV = OSV`" is executed in the user shōen. Its failure can be safely handled by the shōen mechanism.

The top level of the PIMOS with this protection filter mechanism will be as follows.

```
boot :-
  pimos(OS),
  execute((user(S), filter(S,OS)), ...).
```

As the protection filter is inserted automatically, the user program may not be aware of the existence of such a filter, as far as it is properly using the communication stream.

Note that, the two occurrences of the variable `S` in the above program must be unified prior to the invocation of the shōen. Otherwise, this unification may fail. The proper unification order is guaranteed by the rules described in 2.2.5.

6.5 Protocol Compiler

A disadvantage of the protection filter scheme is that the filter must know all the details of the message protocol. It may be common in conventional operating systems that the interfacing code knows all the details of the communication protocol. It may, however, make the system maintenance cost considerably higher because the code of the protection filters for various devices may be lengthy, taking a large part of the PIMOS which is relatively compact.

Fortunately, given the communication protocol of the PIMOS and the user programs, the code for the protection filters can be generated automatically. This generator program is called the *protocol compiler*. Using the

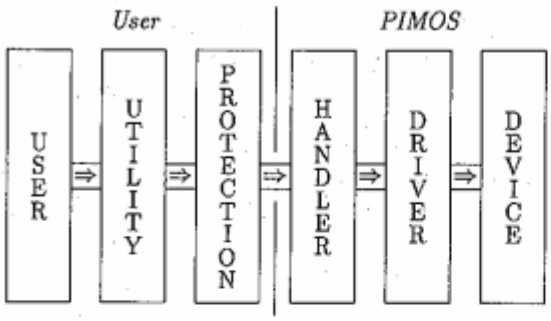


Figure 8: Communication Filters

protocol compiler, a compact specification of the communication protocol with fair readability can be used as the source code of the PIMOS. It is also possible to generate the device handler code using the same technique.[23]

6.6 Summary of Communication

The communication path from a user program to a physical device is summarized in the figure 8. Although I/O messages goes through several filters as shown in the figure, I/O requests are buffered at the utility filter which comes the first, keeping the communication cost to a reasonable level. Also note that all the filters can be executed in a pipelined manner.

6.7 Allocation of Filters

The current version of the PIMOS lets the user determine all the job allocation. All the filters in the communication path from the end user to the device driver are allocated initially to the processor where the user program first requested for the communication path. This makes the communication cost minimum as far as the user program stays in the same processor. The user program can make filters migrate to other processors by sending a reallocation message to the stream afterwards. Filters will not migrate automatically following the user process, for it may not be desirable. The user program may move among processors often but actual I/O may be required only after many migrations, in which case only one migration of filters before massive I/O requests is desirable.

Allocation problem of the processes of the PIMOS should be solved someday with the general load balancing problem, which is one of the most important research topic in the future.

[23] In the version of the PIMOS available when this document is being prepared, all the protection filter and device handler codes are hand-written yet.

7 DEVELOPMENT ENVIRONMENT

This section describes the programming environment provided for the development of PIMOS. Some of the utilities described here may be also useful in development of application programs.

7.1 PDSS and Micro-PIMOS

Prior to the development of the KL1 language implementations on parallel inference machines, an implementation of the language for conventional computers was made in the language C. In addition to the language implementation, a primitive operating system is also built upon it. The system is called the PIMOS Development Support System, or PDSS in short, and the operating system is called the Micro-PIMOS (Miyazaki *et al.* 1988). The primary objective of the development of the PDSS was to provide a program development environment for the KL1 language that enables the development of the PIMOS in parallel with the development of the parallel inference machines.

The PDSS is a pseudo-parallel implementation of the language. Although real parallelism is not in the implementation, all other essential features of the KL1 language are provided by the system. The PDSS implementation thus also played a role of a prototype for implementations on the parallel inference systems. Many of the compilation and other fundamental implementation ideas developed for the KL1 language were verified through this implementation.

The KL1 extensions to the FGHC language were actually utilized in the Micro-PIMOS. For example, the shōen feature and the priority management feature were extensively used in the Micro-PIMOS and have been proved to be effective. The same applies to the various optimizations in the implementation level.

The PDSS system provides various debugging features including the following.

- Reduction by reduction stepwise execution with symbolic trace output. Selective tracing somewhat similar to Prolog debuggers including the *spying* feature is also provided.
- Deadlock detection feature during program execution based upon multiple reference management and during garbage collection. A goal is recognized as deadlocked when it is found to be waiting for a variable to which no other goals have access paths.
- Execution profiling feature counting numbers of invocations of predicates. Useful for program tuning.
- Static program checkers such as void variable detection and predicate dependency analysis.

The system has been revised many times on need adding new features during the development of the PIMOS. These debugging features accelerated the development considerably.

As the parallelism, the only crucial difference between the PDSS and parallel implementations, is implicit on principle in the KL1 language, transporting of the PIMOS from PDSS to the Multi-PSI implementation was extremely easy. As was expected, almost no software synchronization problem was found. This was the greatest merit of writing the system in a logic-based concurrent programming language.

7.2 KL1 Compiler

The compiler for the KL1 language was also written first for the PDSS. As the PDSS implementation and implementations for real parallel machines are both based on the KL1-B abstract machine (Kimura and Chikayama 1987), basically the same compiler could be used for both of them. For parallel implementations, several features were added for enabling parallel execution, but it was quite easy.

7.3 Pseudo Multi-PSI

The Pseudo Multi-PSI is an implementation of the KL1 language on single-processor PSI-II machines (Nakashima and Nakajima 1987). The PSI-II machine is a logic programming workstation, developed earlier in the FGCS project, which is also used as the front-end processor for the Multi-PSI and probably for other parallel inference systems. The processor of the PSI-II is also used as the processors of the Multi-PSI machine.

A pseudo-parallel system for the KL1 language is implemented on the PSI-II by storing two distinct set of microcodes in the microprogram storage: One required for KL1 and the other for KL0, which is the machine language for a sequential logic programming language ESP (Chikayama 1984), originally used in the PSI machine.[24] As the same hardware and the firmware are used, the pseudo Multi-PSI attains almost the same performance as the Multi-PSI system consisting of only one processor.

One *processor* of the Multi-PSI is emulated by one *process* on the *pseudo* version. Pseudo processors are switched when given number of reductions are completed. The scheduler can specify arbitrary scheduling, including random ones. An advantage of pseudo-parallel implementations to real parallel implementations is that the same execution sequence is reproducible. Even if the scheduling is random, it is only pseudo-random; giving the same seed, the same random number sequence can be obtained. This makes bug locating much easier. Symbolic tracing feature of the PDSS was also made available on the pseudo and real Multi-PSI.

[24] In the actual implementation, two versions of the microcodes are overlayed due to lack of storage capacity.

The PIMOS was first transported from the PDSS to this Pseudo Multi-PSI and then to the *real* Multi-PSI. As the PSI-II hardware is much more compact and inexpensive than the Multi-PSI, debugging of many parts of the PIMOS and, more importantly, debugging of the firmware could be carried out in parallel. The same would apply to future improvements of the PIMOS and the language implementation, and also to development of application programs.

8 CONCLUSION AND FUTURE RESEARCH PLANS

The development of the PIMOS showed not only the feasibility but also advantages of using concurrent logic programming languages as the basis of operating systems.

As we already experienced during the development of the SIMPOS (the operating system for the logic programming workstation PSI) in a earlier stage of the project, there are various merits in using a symbolic programming language for description of an operating system (Chikayama 1988). The same merits were observed also during the development the PIMOS. The source programs could be written in a quite readable form. An interactive symbolic debugger was made available from the earliest stage of the development, providing trace output quite easily compared with the source program; without this symbolic trace, the development would have been much more toilsome.

The most notable observation made during the development of the PIMOS was that almost no synchronization problem was found in the debugging phase. Using conventional procedural languages, data-flow synchronization must be transformed into control-flow synchronization by the programmers, which is the largest source of bugs in development of operating systems. Using a concurrent logic programming language, the system designers have to be aware only of the flow of data and almost nothing of synchronization, as data-flow synchronization is implicit in the language. This was the largest merit of using the KL1 language.

There are several problems which are yet to be solved in future research.

One of the most important problems left unsolved is balancing computational load of processors. In the current version of the PIMOS, load balancing is specified by the user programs and the PIMOS merely faithfully obeys that (see section 6.7). A semi-automatic load balancing scheme was proposed (Chikayama 1986) and the basic hardware mechanism required for the scheme was provided (Takeda *et al.* 1988), but it is not utilized in the current version of the PIMOS and its effectiveness has yet to be evaluated.

Another problem left is in the memory management.

The garbage collection mechanism and the quantitative memory management scheme proposed in this paper do not necessarily go together well. A new model of memory consumption may be required here.

In the current version of the PIMOS on the Multi-PSI machine, the front-end processor provides a high-level I/O interface. This scheme gives clean semantics to the physical I/O, but it may be laying too much burden on the back of the front-end processor. A decent model of lower-level I/O operations, which also can be implemented more efficiently, is desirable.

Tuning of various parameters of the PIMOS, communication buffer size or resource management accuracy, for example, are not carried out yet. Such parameters should be determined through future experiences with the system and parallel application software.

ACKNOWLEDGMENTS

Many researchers of ICOT and other related research groups, too numerous to be listed here, participated in the design and implementation of the the KL1 language, the operating system itself and the development tools. We would also like to express our thanks to Dr. S. Uchida, the manager of the fourth research laboratory of ICOT, and Dr. K. Fuchi, the director of the ICOT research center, for their valuable suggestions and encouragement.

REFERENCES

(Chikayama 1984) T. Chikayama. Unique Features of ESP. In *Proceedings of FGCS'84*, pages 292–298. ICOT, Tokyo, 1984.

(Chikayama 1986) T. Chikayama. Load balancing in a very large scale multi-processor system. In *Proceedings of Fourth Japanese-Swedish Workshop on Fifth Generation Computer Systems*. SICS, Stockholm, 1986. Also as ICOT Technical Memorandum, TM-276, ICOT, 1986.

(Chikayama 1988) T. Chikayama. Programming in ESP — experiences with SIMPOS. In K. Fuchi and M. Nivat, editor, *Programming of Future Generation Computers*, pages 75–86. North-Holland, New York, New York, 1988.

(Chikayama and Kimura 1988) T. Chikayama and Y. Kimura. Multiple reference management in flat GHC. In *Proceedings of Fourth International Conference on Logic Programming*, volume 2, pages 276–293. The MIT Press, Cambridge, Massachusetts, 1987.

(Clark and Gregory 1984) K. Clark and S. Gregory. Notes on systems programming in Parlog. In *Proceedings of FGCS'84*, pages 299–306. ICOT, Tokyo, 1984.

(Clark and Gregory 1986) K. L. Clark and S. Gregory. Parlog: A parallel logic programming language. *ACM Transaction on Programming Languages and Systems*, 8(1), 1986.

(Foster 1987) I. Foster. Logic operating systems: Design issues. *Proceedings of the Fourth International Conference on Logic Programming*, volume 2, pages 910–926. The MIT Press, Cambridge, Massachusetts, 1987.

(Foster 1988) I. Foster. Parlog as a systems programming language. *Ph. D. Thesis*, Imperial College, London, 1988.

(Goto *et al.* 1988) A. Goto *et al.*. Overview of the parallel inference machine architecture (PIM). In *Proceedings of FGCS'88*. ICOT, Tokyo, 1988.

(Hircsh *et al.* 1987) M. Hircsh *et al.*. Computation control and protection in the Logix system. In E. Shapiro, editor, *Concurrent Prolog: Collected Papers*, volume 2, pages 28–45. The MIT Press, Cambridge, Massachusetts, 1987.

(Kahn *et al.* 1986) K. Kahn *et al.*. Objects in concurrent logic programming languages. *Sigplan Notices*, 21(11):242–257, 1986.

(Kimura and Chikayama 1987) Y. Kimura and T. Chikayama. An abstract KL1 machine and its instruction set. In *Proceedings of 1987 Symposium on Logic Programming*, pages 468–477. Computer Society Press of the IEEE, Washingon, D.C., 1987.

(Miyazaki 1988) T. Miyazaki. Parallel logic programming language KL1 — Its implementation and an operating system in it —. In *The Transaction of the Institute of Electronics, Information and Communication Engineers*, J71-D(8):1423–1432, 1988. *in Japanese*.

(Miyazaki *et al.* 1988) T. Miyazaki *et al.*. PDSS Manual. ICOT Technical Memorandum, TM-437, ICOT, 1988. *in Japanese*.

(Nakashima and Nakajima 1987) H. Nakashima and K. Nakajima. Hardware architecture of the sequential inference machine PSI-II. In *Proceedings of 1987 Symposium on Logic Programming*, pages 104–113. Computer Society Press of the IEEE, Washingon, D.C., 1987.

(Shapiro 1983) E. Shapiro. A subset of Concurrent Prolog and its interpreter. ICOT Technical Report TR-003, ICOT, 1983.

(Shapiro 1984) E. Shapiro. Systems programming in Concurrent Prolog. In *Proceedings of the 11th ACM Symposium on Principles of Programming Languages*, 1984.

(Shapiro and Takeuchi 1983) E. Shapiro and A. Takeuchi. Object Oriented Programming in Concurrent Prolog. ICOT Technical Report TR-004, ICOT, 1983. Also in *New Generation Computing*, 1-1, 1983.

(Steele 1984) Guy Steele *et al.*. *Common Lisp, the Language*. Digital Press, 1984.

(Takeda *et al.* 1988) Y. Takeda *et al.*. A load balancing mechanism for large scale multiprocessor systems and its implementation. In *Proceedings of FGCS'88*. ICOT, Tokyo, 1988.

(Ueda 1986) K. Ueda. Guarded horn clauses: A parallel logic programming language with the concept of a guard. ICOT Technical Report TR-208, ICOT, Tokyo, 1986.

(Ueda and Chikayama 1984) K. Ueda and T. Chikayama. Efficient stream/array processing in logic programming languages. In *Proceedings of FGCS'84*, pages 317–326. ICOT, Tokyo, 1984.

(Yoshida and Chikayama 1988) K. Yoshida and T. Chikayama. A'UM — a stream-based concurent object-oriented language. In *Proceedings of FGCS'88*. ICOT, Tokyo, 1988.