ICOT Technical Report: TR-246

TR-246

An Abstract KL1 Machine and Its Instruction Set

Y. Kimura and T. Chikayama

June, 1987

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

AN ABSTRACT KL1 MACHINE AND ITS INSTRUCTION SET

Yasunori KIMURA and Takashi CHIKAYAMA

Institute for New Generation Computer Technology
4-28, Mita 1-chome, Minato-ku, Tokyo 108 JAPAN

Abstract

Two parallel inference machine architectures are being investigated at ICOT. One is the *Multi-PSI*[7] and the other is the *PIM*[4]. The target language for both is the parallel logic programming language, KL1, which is based on GHC[8]. Programs in KL1 are compiled to the KL1 instruction set, called *KL1-B*, and executed on the target machine.

To build an efficient multi-processor system, execution on each processing element must be as efficient as possible. Therefore, we have designed an abstract KL1 machine and its instruction set which can execute KL1 efficiently on a single processor. This paper describes the abstract KL1 machine and its instruction set with a compilation scheme to generate efficient codes, concentrating on the sequential execution of KL1, which gives a basis for implementations of multi-processor systems such as the PIM or Multi-PSI. The abstract machine is what we call a *register machine*, which treats *tagged*-data. The instruction set consists of unification, goal manipulation and suspension instructions. This instruction set can also easily be mapped onto general purpose machines as well as onto the PIM and the Multi-PSI.

Finally, this abstract machine is compared with Levy's machine and WAM.

1 Introduction

Two parallel inference machine architectures are being investigated at ICOT. One is the *Multi-PSI*[7] and the other is the *PIM*[3][4].

The Multi-PSI system consists of 16 to 64 PSI-IIs[6] as its processing elements (PEs). They are connected with a two-dimensional mesh type network. Each node of this network has five input/output channels, one of which is connected to the PSI-II's internal bus. Its basic data transfer mechanism is packet switching, and is controlled by the PSI-II's microprogram. This system is the workbench for studying parallel software systems.

The PIM has a hierarchical structure with a cluster concept. Each cluster consists of eight or more PEs which communicate through a shared memory (SM) with a parallel cache system. Each PIM's PE is intended to design using the VLSI technology. For the connection among the clusters, a packet switching network has been adopted although its details have not yet been decided. More than ten clusters are connected with this network as a PIM.

The target language of these systems is the parallel logic programming language, KL1, which is based on GHC[8] and has various features to support the PIM Operating System (PIMOS). The instruction set is called *KL1-B*[1], and corresponds to the *machine instruction set* of conventional machines. Programs in KL1 are compiled to KL1-B code and then executed on the target machine.

To build an efficient multi-processor system, execution on each processing element must be as efficient as possible in the first place. Therefore, as the basis of the parallel architecture research, we have designed an abstract KL1 machine and its instruction set which can execute KL1 efficiently on a single processor.

This paper describes the abstract KL1 machine and its instruction set with a compilation scheme to generate efficient KL1-B codes, concentrating on the sequential execution of KL1 within one processing element. Instructions and functions for inter-processor communication are not described. Therefore, the abstract machine and its instruction set described in this paper give a sequential subset for multi-processor systems such as the PIM or Multi-PSI.

Section 2 gives a brief overview of KL1. Section 3 describes an abstract KL1 machine. Section 4 presents an abstract KL1 instruction set. Section 5 describes the difference between full GHC and flat GHC from the implementation point of view. Finally, section 6 compares KL1, the abstract KL1 machine and the abstract KL1 instruction set, with WAM[9].

2 Overview of KL1

2.1 Language features of KL1

KL1, the parallel logic programming language based on GHC[8], is one of the committed choice languages.

A GHC program is a finite set of guarded Horn clauses in the following form:

$$H : -G_1, \ldots, G_m | B_1, \ldots, B_n. (m \geq 0, n \geq 0)$$

[1] 'B' is an abbreviation for 'base'

where H, G_i's, B_j's are called the head, the guard goals and the body goals respectively, and '|' is called the commitment operator. The part of a clause preceding '|' is called the passive part and that following it is called the active part.

When input goal H is given, reduction of H is tried *in parallel*, and a clause whose head unification and guard goal execution succeeded first is selected. After that, body goals B_j's are executed. This means that goal H is reduced to B_j's. If unification requires the instantiation of a variable during passive part execution, this unification is suspended.

Taking the efficient implementation into consideration, *flat* GHC was adopted as KL1. Flat GHC is a subset of GHC which allows only built-in predicates as guard goals. This restriction enables more efficient implementation because a single environment suffices for this specification, and almost all the GHC programs can be translated into flat GHC without essential changes.

The implementation of *full* GHC is discussed in [5].

[8] gives a more detailed description of the specification of the language GHC.

2.2 Execution mechanism of KL1

To show the execution mechanism of KL1, consider the next example:

```
?- p(X), q(X).                                  (1)

p(X) :- true    | X=[a|Y], p(Y).               (2)
q(X) :- X=[a|Y] | q(Y).                        (3)
q(X) :- X=[b|Y] | q(Y).                        (4)
```

Assuming that two goals p(X) and q(X) are given, they are stored in a *goal pool*. The goal pool is used for maintaining the goals. Then, reductions of p(X) and q(X) start *in parallel*. That is, the reduction of goal p(X) with clause (2) is tried and that of goal q(X) with clauses (3) and (4) is also tried. Clause (2) is selected immediately and the new goal, p(Y), is generated after instantiating X to [a|Y]. In other words, goal p(X) is reduced to goal p(Y), and this new goal is put in the goal pool in place of the original p(X).

When the reduction of q(X) is tried, the unification between X and a list ([a|Y] or [b|Y]) is tried. If X is unbound, the unification does not succeed and the execution of goal q(X) is suspended, waiting for X to be instantiated. A goal such as q(X) is called a *suspended goal*. Suspended goals can resume execution when the variable is instantiated by some concrete value. In the above example, the other goal p(X) binds X to [a|Y]. Therefore, the reduction of q(X) is eventually resumed. Then the goal q(Y) is put in the goal pool in place of q(X).

The execution of KL1 programs proceeds by repeating these operations.

2.3 Sequential implementation of KL1

To build an efficient multi-processor system, execution on each processing element must be as efficient as possible. Therefore, an abstract KL1 machine and its instruction set based on a sequential execution is considered in the first place. To extend the sequential system to a multi-processor system with a shared memory, it is necessary to treat goals and variables with exclusive memory access [2].

The following items should be considered when implementing KL1 on a sequential processor.

- Manipulation of goals

  As in the above example, the possible states of goals can be classified as follows:

  1. *Ready goal*: Goal ready for execution in the goal pool.
  2. *Suspended goal*: Goal which is suspended, waiting for some variable to be instantiated.
  3. *Current goal*: Goal *currently* being executed.

  The set of ready goals in the goal pool are represented as the *ready queue*. The ready queue can be realized by linking the records that represent the goals with *one*-directional pointers. Getting goals from or returning goals to the goal pool corresponds to manipulating the ready queue.

- Execution of the passive part

  Each candidate clause is tested sequentially by head unification and guard execution in order to choose one clause whose body goals will be executed. If instantiation of a variable is required during the execution of the passive part, the test for this clause is abandoned and execution proceeds to the next candidate clause. The clause whose head unification and guard goal execution succeeded first is selected. If no clause is selected, execution of that goal is suspended. This is called *suspension* and the manipulation of suspension is called *suspension processing*.

- Execution of the body part

  If a clause is selected, the body part of that clause is executed. Execution of the body part consists of two operations, unification (*active unification*) and *body goal fork*. Active unification is executed on the spot, and suspended goals may be resumed by active unification. This is called *resumption* and the manipulation of resumption is called *resumption processing*. The body goal fork is realized by linking goal records to the ready queue.

- Suspension and resumption of goals.

  When execution of a goal is suspended, its goal record is linked from the variable which caused the

suspension, possibly with other goals which have also been suspended on the same variable. Here, the non-busy waiting method has been adopted for *suspension processing*.

When a suspended goal is resumed, its goal record is linked again to the ready queue. Note that *resumption* of a goal does not necessarily mean immediate execution of that goal.

3 Abstract KL1 machine

This section describes the abstract KL1 machine in detail.

3.1 Data representation

Data objects treated by the abstract KL1 machine consist of tags and values. General structures other than lists are not described in this paper, but it is easy to extend the list manipulation treatments to general structures.

The tag can have one of the following values:

`UNDF`: Uninstantiated variable. The value has no meaning.

`HOOK`: Uninstantiated variable, some goals are waiting for instantiation of this variable. The value is the pointer to these goals. These goals are said to be *hooked* to the variable.

`REF` : Reference pointer to a variable cell either uninstantiated or *hooked*.

`INT` : Integer. The value is the integer value itself.

`ATOM`: Symbolic atom. The value is an atom identifier.

`LIST`: List cell. The value is the pointer to the consecutive two words representing the car and cdr parts of the list cell.

3.2 Structures

A data structure called a *goal record* is used for representing a goal in the abstract machine. The ready queue is realized by linking such goal records.

- Goal record

  A goal record consists of the arguments of the goal and its execution environment. Individual fields of the record are:

  Narg: number of arguments of this goal;

  $argument_0$: first argument;

  $argument_{Narg-1}$: Narg-th argument;

  code: address of the predicate code;

  next goal record: pointer to the next goal record in the ready queue.

- Ready queue

  The ready queue is implemented by linking the ready goal records with their *next goal record* fields. In the current implementation, the ready queue is maintained as a *stack* rather than a *queue*.

To implement the suspension and resumption, the following structures are used.

- Suspension record

  A suspension record records the goal records suspended on a variable, and consists of two fields.

  next suspension record: pointer to the next suspension record. If more than one goal record is suspended on the same variable, the suspension records for them are linked by this field.

  suspension flag record: pointer to the suspension flag record.

- Suspension flag record

  This is required to allow multiple waiting, i.e., one goal waiting for instantiation of one of several variables, and consists of two fields.

  goal record: pointer to the suspended goal record.

  suspension count: number of variables the goal is suspending.

- Suspension stack

  This stack is a working stack that preserves the variable which may cause the suspension of the goal temporarily while candidate clauses for a goal are being tried. This stack is cleared when some clause is selected or suspension processing is completed.

The latter part of this sub-section shows how multiple waiting is manipulated. Consider the following example.

```
?- p(X, Y), q(X),...

p([a|X1], Y) :- true | p(X1, Y).
p(X, [a|Y1]) :- true | p(X, Y1).
q([a|X1]) :- true | q(X1).
```

Figure 1 shows an example of multiple waiting. The goal, p, is hooked on variables X and Y, and the other goal, q, is hooked only on variable X. The suspension count field of suspension flag for p is *2* and that for q is *1*. Assume that goal p is resumed by the instantiation of variable Y . The goal record field of the suspension flag

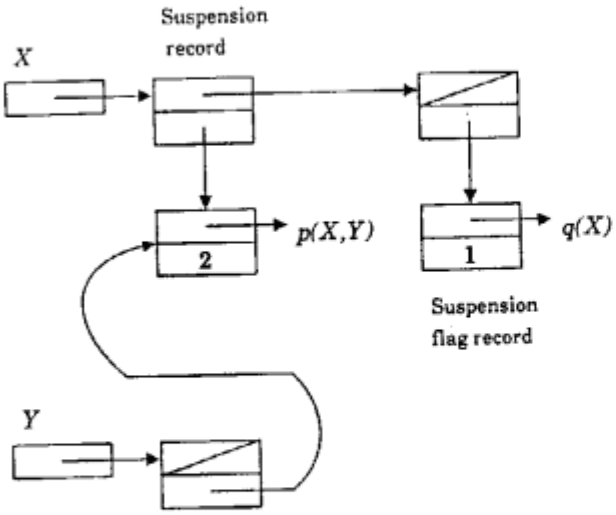

Figure 1: Before

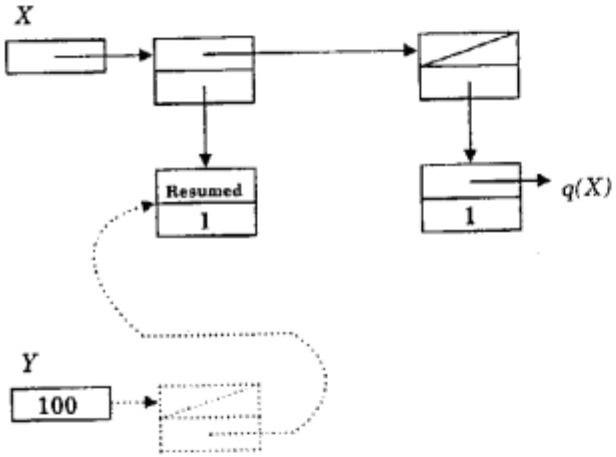

Figure 2: After

record for p is replaced with some special value showing that the goal has already been resumed, and the suspension count is decremented by one, as shown in Figure 2.

3.3 Registers

Several registers are used for maintaining the *current goal* and for managing the memory resource.

It is assumed that the contents of the current goal record are loaded to these registers before its reduction begins.

Structures such as goal records, suspension records and suspension flag records are allocated in the memory area, and are maintained by corresponding *free lists*. Variable cells are allocated in the garbage collected heap. The suspension stack is allocated at a fixed memory area with a fixed size.

PC: program counter;

Carg: number of arguments of the current goal;

$argreg_0$: first argument;

$argreg_{Carg-1}$: Carg-th argument;

SR: structure pointer used for the structure unification;

HP: heap top pointer;

SSP: suspension stack top pointer;

RQHP: pointer to the ready queue head;

GRFP: pointer to the free goal record list;

SRFP: pointer to the free suspension record list;

SFFP: pointer to the free suspension flag record list;

write_mode: mode flag used for structure unification.

4 Abstract Instruction Set

The instruction set and its compiler are designed according to the following principles.

- Most passive part instructions have branch labels, each of which indicates the code address to be tested next when they are suspended.
- After dereferencing the value of an instantiated variable, the dereferenced result is put back where the reference pointer originally was.
- The argument registers are never destroyed before some clause is selected, except for being replaced by the dereferenced value.
- Suspension never occurs during execution of built-in predicates in the guard.
- One of the body goals in the selected clause is executed *tail recursively*, and others are linked to the ready queue.
- The instruction for the *suspension processing* is explicitly generated.

The KL1 instruction set is roughly classified into six groups. Note that the instructions for general structures are omitted in the later section, because they are similar to that of list.

1. Passive unification instructions

2. Guard part built-in predicates
3. Active unification instructions
4. Goal manipulation instructions
5. Argument preparation instructions
6. Suspension instruction

4.1 Passive unification instructions

Passive unification instructions, shown below, are used in the passive part unification.

```
wait_variable Xj, Ai
wait_value Xj, Ai, Label
wait_constant C, Ai, Label
wait_list Ai, Label
read_variable Xj, Label
read_value Xj, Label
read_constant C, Label
```

In these instructions, Ai is an argument register and Xj is a temporary register used as the work register.

Passive unification may require instantiation of variables to accomplish the unification, in which case the unification should be suspended. Additionally, they may fail. Therefore, the branch address, *Label*, is provided in each instruction which may fail or suspend to jump to the next alternative clause code or a *suspend* instruction.

In these instructions, the dereferencing[2] of the argument register is performed first and the result is put back in the register where the reference pointer originally was. This is effective in avoiding duplicated redundant dereferencing of the same variable in the passive part.

For example, *wait_constant C, Ai, Label* can be defined and used as follows:

- Definition

```
Wait_constant(C, Ai, Label):
  put the dereference result of Ai to Ai
  check the equality between Ai and C
  if they are equal then proceed to the next code
  elseif Ai is uninstantiated
    then push Ai to the suspension stack and
      jump to Label
    else jump to Label
```

- KL1 sample program

```
p(foo, Y) :- true | true.
p(bar, Y) :- true | true.
```

- Compiled code

```
p/2:    wait_constant 'foo', A1, p/2/1
        proceed
p/2/1:  wait_constant 'bar', A1, p/2/2
        proceed
p/2/2:  suspend p/2
```

An optimizing compiler can generate code which reduces the number of guard tests. For example,

```
p([_|_], foo) :- true | true.        (1)
p([],    bar) :- true | true.        (2)
```

are compiled to

```
p/2:    wait_list A1, p/2/1
        wait_constant 'foo', A2, p/2/2
        proceed
p/2/1:  wait_constant [], A1, p/2/2
        wait_constant 'bar', A2, p/2/2
        proceed
p/2/2:  suspend p/2
```

If the first argument of p/2 is a list, clause (2) is never selected. Therefore, the label of *wait_constant foo A2* directly points the suspension instruction. This is why *each* passive unification instruction has *Label*.

4.2 Built-in predicates

The passive part of a clause may contain calls of built-in predicates. Some of the instructions for built-in predicates are:

```
integer  Ai, Label      wait  Ai, Label
equal  Ai, Aj, Label    greater  Ai, Aj, Label
add  Ai, Aj, Ak         multiply  Ai, Aj, Ak
modulo  Ai, Aj, Ak
```

When the built-in predicates are called, their input variables must be instantiated to the legal values to avoid the suspension within its body routine. To accomplish this, the instantiation and data type of the input variable of predicates are checked out by *integer* and *wait* instructions.

Integer Ai, Label dereferences Ai and puts the value in Ai, then checks whether the result is an integer. If the result is an integer, the execution proceeds to the next instruction; otherwise, it jumps to the address indicated by *Label*. Consider the example:

- KL1 source program

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

[2]The dereferencing Ai returns the value itself if Ai is instantiated, otherwise the reference pointer to the uninstantiated variable is returned.

- Compiled code

```
p/2:    integer A1, p/2/1
        integer A2, p/2/1
        greater A1, A2, p/2/1
        proceed
p/2/1:
```

4.3 Active unification instructions

Active unification instructions, shown below, are used for active part unification.

```
get_variable Xj, Ai     get_value  Xj, Ai
get_constant C, Ai      get_list   Ai
unify_variable Xj       unify_value  Xj
unify_constant C
```

Active unification instructions are never suspended. Therefore, if instantiation of the hooked variable is required, the goals hooked on that variable are resumed. This processing is done on the spot within each instruction. The *mode* is used for the structure unification. The operations taken by *unify* instructions depend on this *mode* as in [9].

For example, *get_list Ai* is defined and used as follows:

- Definition[3]

```
Get_list(Ai):
  put the dereference result of Ai to Ai
  if Ai is uninstantiated
    then Ai := list!HP and write_mode := ON
      proceed to the next code
  elseif Ai is list
    then SR := ref!Ai and write_mode := OFF
      proceed to the next code
    else jump to Fail
```

- KL1 source program

```
p([A|X], Y) :- true | Y = [A|Y1], p(X, Y1).
```

- Compiled code[4]

```
p/2:
  wait_list Ai, p/2/1       % p([
  read_variable X3          %   A|
  read_variable A1          %   X], Y) :- true |
  get_list A2             *1 % Y = [
  unify_value X3          *2 %     A|
  unify_variable A2       *3 %      Y1],
  execute p/2               % p(X, Y1).
p/2/1:
  suspend p/2
```

4.4 Goal manipulation instructions

```
create_goal   Code, Arity
enqueue_goal  Goal/Arity
execute       Code
proceed
```

The *create_goal Code, Arity* instruction allocates a new goal record whose code address and arity are *Code* and *Arity* from the *free goal record list* and *enqueue_goal Goal/Arity* links the new goal record to the ready queue. (The operand, Goal/Arity, is only for readability of the code.) These two instructions are always used as a pair, and the argument set instructions are sandwiched between them. This may seem redundant but is advantageous for simplicity. This is how a *goal forks*.

The *execute Code* instruction moves the execution control to the code address indicated by *Code*. This is used for *tail recursive* execution[5].

The *proceed* instruction is used, when one goal is reduced completely, for getting a new goal record from the ready queue. Then it loads the arguments in the record to the argument registers, restores the execution environment, and jumps to the code address. For example, *create_goal Code, Arity* is defined as follows:

```
Create_Goal(Code, Arity):
  get a new goal record New_record
      from the free goal record list
  put Arity to the Narg field of New_record
  put Code to the code field of New_record
  proceed to the next code
```

4.5 Argument preparation instructions

Argument preparation instructions, shown below, are used for preparing the arguments of a goal record to be forked.

```
put_variable Xj, Ai    set_variable Xj, Gi
put_value Xj, Ai       set_value Xj, Gi
```

[3] Ai:=list!HP means to put the value of HP with its tag *list* to Ai.

[4] Instructions marked with *1, *2 and *3 are discussed in section 4.7

[5] Current implementation executes the leftmost goal of the body *tail recursively*.

```
put_constant C, Ai      set_constant C, Gi
put_list Ai             set_list Gi
write_variable Ai
write_value Ai
write_constant C
```

They are classified into three categories according to the destination to which the arguments are saved.

Set_XXX instructions save the argument value or variable in the argument save area of a goal record. Gj of the *set_XXX* instructions denotes the j-th slot of the goal record. *Put_XXX* instructions are used for re-arranging the contents of registers prior to the tail recursive execution of a goal. *Write_XXX* instructions are used for putting a structure element. Note that they are always executed in *write* mode.

For example, *Put_list Ai* and *write_value Ai* are defined and used as follows:

- Definition

```
Put_list(Ai):
  Ai := list!HP
  proceed to the next code

Write_value(Ai):
  (HP) := Ai
  increment HP by the word length
    of the variable cell
  proceed to the next code
```

- KL1 source program

```
p(X, Y, Z) :- true | q(Y, Z),
              r([X|A]), s(A, Z).
```

- Compiled code

```
p/3:
  create_goal s, 2        % s(
  set_variable X4, G1     %  A,
  set_value A3, G2        %    Z)
  enqueue_goal s/2
  create_goal r, 1        %  r(
  set_list G1             %    [
  write_value A1          %     X|
  write_value X4          %       A])
  enqueue_goal r/1
  put_value A2, A1        %     q(Y,
  put_value A3, A2        %         Z)
  execute q/2
```

4.6 Suspension instruction

```
suspend  Goal/Arity
```

The *suspend Goal/Arity* instruction is used for suspension processing.

Suspend Goal/Arity is roughly defined as follows:

```
Suspend(Goal/Arity):
  if the suspension stack is empty
    then goto Fail
  while the suspension stack is not empty
    Var := (SSP) and decrement SSP by one word.
    if Goal is already hooked on Var
      then do nothing
    else get a new suspension record Susp_record
        from free suspension record list
      link Var to Susp_record
      if the suspension flag record for Goal
        is already allocated
        then increment the suspension count of
          the suspension flag by one
      else
        get a new suspension flag record Susp_flag
          from free suspension flag record list
        put Goal to the goal record field
          of Susp_flag and
        set 1 to its suspension count
```

4.7 Optimization

4.7.1 Register allocation

In KL1, the guard goals are all built-in predicates and their arguments are loaded to the argument registers before reduction in this implementation. This means that the execution of guard built-in predicates proceeds by calling its body routine after arranging the arguments in the registers in turn. During execution, although the arguments passed from the calling goal must be preserved before commitment, the arguments can be moved and the registers can also be reused from the point in the guard where the compiler recognizes the clause is committed.

Consider the following example:

- KL1 source program

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

- Compiled code without optimization

```
p/2:
  integer A1, p/2/1          % X is integer ?
  integer A2, p/2/1          % Y is integer ?
  greater A1, A2, p/2/1      % X > Y |
  add A1, A2, X3             % Z := X + Y |
  put_value X3, A1
  execute q/1                % q(Z)
p/2/1:                       % Next Clause
```

- Compiled code with optimization

```
p/2:
  integer A1, p/2/1          % X is integer ?
  integer A2, p/2/1          % Y is integer ?
  greater A1, A2, p/2/1      % X > Y |
  add A1, A2, A1             % Z := X + Y |
  execute q/1                % q(Z)
p/2/1:                       % Next Clause
```

Without register allocation optimization, the result of adding X and Y is put in the X3 register. However, this built-in predicate,add, never causes suspension and is the last predicate of this guard part. Therefore, register A1 can be used as both the output register of 'add' and the register for the first argument of q/1, because the contents of A1 and A2 are never accessed after being passed to the body routine of 'add'.

4.7.2 Active unification of a variable and a structure

As described in 4.3, the instructions using *mode* are generated for unifying a variable with a structure in the active part. From the multi-processor implementation point of view, such structure unification causes memory locks for a long period. In the compiled example shown in section 4.3, variables A and Y1 must be accessed exclusively after locking the variable Y between *2 and *3. This is because some other processor may attempt to access the head or tail of Y before they are created in *2 and *3. Taking a hardware memory lock mechanism into consideration, memory locks of such a long period will be inconvenient.

Therefore, to shorten the lock period of variables, another compiling scheme is introduced. The previous example in section 4.3 is compiled as follows:

```
p/2:
  wait_list A1, p/2/1          % p([
  read_variable X3             %   A|
  read_variable A1             %   X], Y) :- true |
  put_list X4                *1 % [
  write_value X3             *2 %   A|
  write_variable X5          *3 %     Y1]
  get_value X4, A2           *4 % [A|Y1] = Y,
  put_value X5, A2           *5 % p(X, Y1)
  execute p/2
p/2/1:
  suspend p/2
```

Instructions *1 to *3 create a new list structure, [A|Y1]. Then the instruction, *get_value X4, A2*, unifies Y in the second argument, A2, with the newly created list, X4. Therefore, exclusive memory access is required only while *4 is being executed.

With this compiling scheme, *mode* is no longer required nor are structure unification instructions such as *unify_XXX* required, because a variable and a structure are always unified after constructing the structure.

On the other hand, if the variable Y in Y = [A|Y1] should already be bound to some list[6], it is obvious that there are disadvantages in time and space efficiency. This is because, in *read mode* unification, there is no need to construct a new list ([A|Y1]) and the new list cell becomes *garbage* after the unification of Y and the newly created list. However, according to preliminary experiments, active unification is nearly always executed in *write* mode in WAM terminology; that is, the variable is uninstantiated in most cases. In fact, we have never seen a program which has the *read* mode unification except when written deliberately.

Additionally, *Xj* in *get_value* is always bound to the newly created list in *4. Therefore, by introducing following specialized instruction *get_list_value Xj, Ai*, execution time can be reduced further.

```
Get_list_value(Xj, Ai):
  put the dereference result of Ai to Ai
  if Ai is uninstantiated
    then Ai := Xj and goto the next code
  elseif Ai is list
    then do general unification between Xj and Ai
    else jump to Fail
```

As a result, optimized active unification instructions are as follows:

```
get_variable Xj, Ai    get_value  Xj, Ai
get_constant C, Ai     get_list_value Xj, Ai
```

4.7.3 Indexing

To execute the KL1 program at a high speed, it is very important to detect the suspension. One method is to find what kind of value each clause expects in the compiling time. Therefore, the following simple indexing instruction is introduced,

```
switch_on_term  Ai, Lc, Ll, Lv
```

where Ai is the argument register to be checked, Lc is the branch address to which control is moved when the contents of Ai are a constant value (integer or atom), Ll is the case where Ai is a list, and Lv is the case where Ai is a variable.

In KL1, the order of the tests for selecting a clause is not defined. Therefore, the compiler can rearrange the order of clauses to generate the *indexing* instructions easily. The compiler collects the clauses whose *Ai*'s are of the same data type, and compiles them in bulk. The general

[6] This corresponds to the *read mode* unification in WAM

flow of clause selection using *Switch_on_term Ai, Lc, Ll* and *Lv* is as follows:

```
put the dereference result of Ai to Ai
switch (type of Ai)
  Constant: try clauses which begin from Lc
    if no clause is selected
      then try clauses which begin from Lv
    if no clause is selected again
      then goto Fail
  List: try clauses which begin from Ll
    if no clause is selected
      then try clauses which begin from Lv
    if no clause is selected again
      then goto Fail
  Undf: try clauses which begin from Lv
    if no clause is selected
      then push Ai to the suspension stack and
      jump to the suspension instruction
```

Since KL1 does not have *backtracking* mechanism, clause indexing may not be as effective as in Prolog. However, it is clear that redundant checks in the guard part can be avoided, and the frequency of the break of the *instruction stream* can be reduced. This property is desirable for *pipelined* processing elements.

5 Related work

Jacob Levy of Weizmann Institute of Science proposed another implementation scheme of GHC in [5]. A notable difference between his work and this paper is that his work tries to implement *full* GHC very efficiently, while this paper discusses that of *flat* GHC.

Full GHC permits the programmer to write user defined goals in its guard part while flat GHC does not. Comparing these two implementations focusing on this point, the following is clear.

- It is necessary to manage the separate OR-parallel environments during guard execution in full GHC. To do this, Levy introduced many runtime structures such as environment vector (EV), status word vector (SW) and mutual exclusion ring (ME).

  EV and SW are not required in KL1. ME corresponds to the suspension record and the suspension flag record in this implementation, and both are used to avoid a case where the suspended goal is resumed many times.

- In full GHC, when the variables are unified, a new data type called *cross environment reference* (CER) is used to avoid the instantiation of variables which are not in the committing goals. When the guard execution reaches commitment, all environments belonging to this guard part user goals should be checked to see whether CER's can be modified to *normal* references or not. This appears to require a great deal of work.

  In KL1, this dynamic check is not required because KL1 does not need any multiple environments in guard part execution.

- Suspension may occur in the following two cases in Levy's implementation. One is when an attempt is made to instantiate the goal variables in the guard part, the other is when the goal waits for some guard to complete its computation and to be ready for commitment. Only the former case is applicable to KL1. It implies that suspension occurs more frequently in full GHC than in flat GHC.

From these points, implementation of *full* GHC is rather complicated compared with KL1 (flat GHC). Execution speed seems to be considerably slower than KL1. On the other hand, it is true that programming in full GHC is easier than in KL1 because user defined goals can be written in the guard part. To compensate for it, we have been designing a user language called *KL1-U*(ser) with a concept of *object oriented* or *modular* programming. KL1-U is transformed to KL1 and compiled to KL1-B code. Thus, the user or system programmers can write programs easily with KL1-U and the target machine executes KL1-B code very efficiently.

6 Comparison with WAM

We have designed a KL1 abstract instruction set, KL1-B. It seems that there are many similarities between KL1-B and WAM [9], which was designed by D.H.D. Warren for Prolog implementation. This section discusses their similarities and differences.

The similarities are:

- Both machines are *register machines*, and unification is performed on the data in these registers.
- Optimization by the compiler, such as register allocation, is expected.
- To maintain ready goals, the *ready queue* is used. In the implementation described in this paper, the ready queue is manipulated as a *stack*. That is, a new goal record to be forked is linked to the head of the ready queue and a goal record is dequeued from it. This method is similar to that of the *goal-stacking* model which is Warren's previous implementation of the abstract machine[7].

The differences are:

[7] This can be found in [9].

- In this implementation or goal-stacking model, all goal arguments are *temporal* and can correspond directly to hardware registers, while in WAM, variables are classified as *temporal* or *permanent*.
- Prolog's unification is *two-way*, but KL1's is close to a *one-way* unification. In KL1, passive unification is one-way, and even active unification is nearly always executed in *write* mode, in the sense of Prolog. Therefore, the *write* mode oriented active unification instructions were designed as described in 4.7.2.
- Prolog's execution proceeds using the the various stacks such as *local, global and trail stack*. Therefore, the memory space is reclaimed implicitly by shrinking the stack by backtracking or deterministic tail recursion.

  On the other hand, KL1's execution proceeds using the heap. There is possibility that the memory consumption is much higher than that of Prolog.

  Therefore, it is necessary to maintain the KL1 memory space efficiently. In KL1, goal records and suspension records can be easily managed by free lists because their life time can be definitely found in the KL1-B code. However, it is difficult to detect the life time of the variable cells from the KL1-B code. Considering this insight, we have already proposed a new memory management mechanism called the *MRB* method[1].

It is stated in [9] that in the goal stacking model, the treatment of the *unsafe* variable is a severe problem. In this model, however, this problem does not occur because uninstantiated variable cells are always allocated in the variable cell area in the heap instead of arguments area of goal records. Therefore, variables which are shared between goals are put in the independent area from goal records, and references to uninstantiated variable cells are put in the argument area of goal records. This is because in multiple processor implementation, it is impossible to detect *unsafe* variables in runtime since goals are executed in parallel and execution of some goals may be suspended.

It is clear that this scheme requires extraneous memory space. However, most space can probably be reclaimed by the garbage collection using MRB.

7 Conclusion and future research

The KL1 abstract instruction set with some optimization and its compiler were designed. The compiler is currently written in Prolog, and are going to be re-written in KL1 itself. The emulator which executes these instructions runs on the Balance 21000 system and its execution speed is over 1 KRPS (Reduction Per Second) on each PE. Functions for inter-processor communication are now being investigated.

Acknowledgements

The research and development outlined in this article is being conducted mainly by the members of the PIM and the Multi-PSI groups in the ICOT Research Center and the participating companies. We wish to thank Mr. Koichi Kumon of Fujitsu Laboratories Ltd. who wrote the emulator, Mr. Hiroshi Nakashima of Mitsubishi Electric Corp. who gave us many valuable suggestions in designing the instruction set, Mr. Hajime Shimizu who pointed out the importance of the memory lock mechanism, and Dr. Atsuhiro Goto and Dr. Evan Tick who gave us helpful comments on earlier drafts. We also wish to thank the Director of ICOT, Dr. Kazuhiro Fuchi, and Dr. Shun-ichi Uchida for valuable suggestions and guidance.

References

[1] T. Chikayama and Y. Kimura: Multiple reference management in Flat GHC. In *Proc. of the 4th International Conference on Logic Programming*, May 1987, also to appear as ICOT Technical Report TR 248.

[2] M. Sato et al: KL1 execution model for PIM cluster with shared memory. In *Proc. of the 4th International Conference on Logic Programming*, May 1987, also to appear as ICOT Technical Report TR 250.

[3] S. Uchida: *Toward the Parallel Inference Machine.* ICOT Technical Report TR 196, ICOT, 1986.

[4] A. Goto and S. Uchida: *Toward a High Performance Parallel Inference Machine -The Intermediate Stage Plan of PIM-.* ICOT Technical Report TR 201, ICOT, 1986.

[5] Jacob Levy: A GHC abstract machine and instruction set. In *Proc of the 3rd International Conference on Logic Programming*, July, 1986.

[6] H. Nakashima and K. Nakajima: Hardware Architecture of the Sequential Inference Machine PSI-II. In *Proc. of the 4th Symposium on Logic Programming*, August 1987.

[7] K. Taki: The parallel software research and development tool : Multi-PSI system. In *Proc. of the France-Japan Artificial Intelligence and Computer Science Symposium 86*, October 1986.

[8] K. Ueda: *Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard.* ICOT Technical Report TR 208, ICOT, 1986.

[9] David H.D. Warren: *An Abstract Prolog Instruction Set.* Technical Note 309, Artificial Intelligence Center, SRI, 1983.