ICOT Technical Report: TR-530

TR-530

A Detection Algorithm of Perpetual Suspension in KL1

by
Y. Inamura & S. Onishi

February, 1990

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

A Detection Algorithm of Perpetual Suspension in KL1

Yū Inamura
Satoshi Onishi
Institute for New Generation Computer Technology

Abstract

KL1 is a committed-choice language, designed as the kernel language of the parallel inference machines which are under development in the Japanese fifth generation project. It is known that committed-choice languages are suitable to describe the concurrent processes because synchronization is supported as a language primitive. A producer/consumer model, in which a producer and a consumer cooperate using data-flow synchronization, is a typical programming style of these languages. However, one thing we must treat carefully is that the execution can fall into the state of perpetual suspension because of some mistakes in the programs, in the case of such data-flow computation. Fixing such mistakes would be very difficult if it were not for the implementation supports. Therefore, as a practical tool for the software development, it is extremely important to detect an occurrence of such an illegal state and to notify it to users.

This paper presents an algorithm to detect an occurrence of perpetual suspension and to report the maximal goals from the causality graph of perpetual suspension. The algorithm takes advantage of the characteristics of copying garbage collection scheme, and have already been implemented on the Multi-PSI, a prototype of the parallel inference machines.

```
?- producer(X), consumer(X).

producer(X) :- true | X = [msg|X2], producer(X2).
consumer([msg|X]) :- true | consumer(X).
```

Figure 1: A Simple Producer and Consumer

1 Introduction

KL1 is a committed-choice language based on Flat GHC[11], and is the system and the user language of the parallel inference machines (PIMs)[6], developed in the Japanese fifth generation project.

Like other committed-choice languages such as CP[10], PARLOG[4], and Strand[5], all the AND goals of a KL1 program can be executed in parallel, and the execution is controlled by the data-flow synchronization using shared variables. In a typical programming style of a committed choice language, a *producer/consumer* model, one variable is shared by a producer and a consumer, and the execution of the consumer is synchronized to that of the producer, by letting the consumer wait for the instantiation of the shared variable. *Guard unification* mechanism of the language provides the way to realize this synchronization.

As the synchronization is supported as a language primitive, the programmability of committed-choice languages is much better than that of other parallel languages with which a programmer must take care of the synchronization. For instance, from our experiences in writing an operating system, PIMOS[3], for the Multi-PSI[9] with KL1, there was no mistakes in terms of the synchronization, and this fact proves the excellence of the language.

However, there is one problem in the case of such data-flow computation as KL1 that the execution of a program can be suspended suddenly, because of some mistakes in the program. We show one example of this problem.

Figure 1 shows a simple producer/consumer program written in KL1.

Suppose that the **producer/1** in figure 1 releases the shared variable without instantiating it as follows.

```
producer(X) :- true | Y = [msg|X2], producer(X2).
```

Then the execution of the `consumer/1` is perpetually suspended, because no other goals can resume the execution of the goal `consumer/1` by instantiating that variable. We call this a state of *perpetual suspension*.

Of course, this mistake shown in this example is brought about by a simple mis-typing, and this can be fixed by a static analysis such as a *variable checker*, which examines the number of appearances of each variable in one clause. However, there are sorts of mistakes which are very difficult to detect by static analyses, and therefore, a dynamic detection mechanism is required.

Another problem we must consider is that perpetual suspension is infectious. Once a goal falls into perpetual suspension, also other goals tend to fall into that state, because usually a consumer is, on the other hand, a producer for another consumer. In the case of practical applications, the dependencies between goals are complicated and large number of goals fall into the state of perpetual suspension.

Therefore, programmers must be faced with the sudden stop of their programs and with a large number of goals of perpetually-suspended, although only a few of these goals are the actual causes of perpetual suspension.

This paper presents an algorithm to detect an occurrence of perpetual suspension, which has already been implemented on the Multi-PSI. The feature of this algorithm is that (1) an occurrence of perpetual suspension can be detected during garbage collection, whether other normal goals exist or not, and that (2) only the maximal goals in the causality of perpetual suspension are discovered. The latter characteristic is especially important as a practical debugging tool, because these maximal goals is the actual causes of perpetual suspension. This can greatly decrease the users' efforts, by restricting the number of the suspicious goals.

The rest of this paper is organized as follows; section 2 describes how to report an occurrence of perpetual suspension; section 3 describes the causality and the maximality; section 4 describes the algorithm for the detection of perpetual suspension; section 5 gives some discussions; and section 6 concludes this paper and describes a future work.

2 Report of Perpetual Suspension

2.1 How to Report Perpetual Suspension

There are some requirements about the form of the report which gives information on perpetually-suspended goals.

1. It is not desirable to report all of the perpetually-suspended goals, because only a few goals are the actual causes of perpetual suspension; and

2. enough information to identify the goal must be reported, hence, not only the name of the perpetually-suspended goal but also its arguments are necessary to be reported.

To satisfy the former requirement, the maximal goals[1] in the causality of perpetual suspension are reported. These maximal goals can be regarded as the actual causes of perpetual suspension.

To realize the latter requirement, the exception handling mechanism is utilized as described in section 2.2.

2.2 Exception Handling Mechanism of KL1

When occurrences of perpetual suspension are detected, it is necessary to notify them to users for helping their debugging. Since perpetual suspension can be regarded as an exceptional event, we decided to treat perpetual suspension just like other exceptional events such as *failure*, *unification failure*, and so on. To handle these exceptions, a meta-programming facility, *Shōen*[3], which is introduced as a language primitive, is utilized. Figure 2 shows the logical structure of the shōen mechanism. Goals are executed inside a shōen, and the execution of the shōen is controlled through the control stream. The status of the computation in the shōen is notified through the report stream. A control process of a shōen watches the report stream and send certain messages through the control stream to control the shōen. In other words, the shōen mechanism can be regarded as an interpreter of the KL1 language.

[1] The meaning of the maximality will be defined in the next section

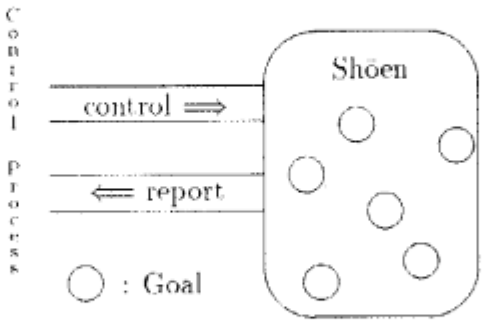

Figure 2: Shōen

Using this Shōen mechanism, an exceptional events such as perpetual suspension is reported by sending a message of the form shown below to the report stream.

```
exception(Info,Goal,NewGoal)
```

The meaning of each argument is as follows.

Info: The reason of the exception.

Goal: Information on the goal which caused the exception, and represented by two terms; a code pointer and an argument vector.

NewGoal: Two variables, for a code pointer and for an argument vector, to specify a goal that will be executed in place of the **Goal**.

The control process which receives the message will treat the exception by instantiating the variables of **NewGoal**.

Note that the perpetually-suspended goals other than the maximal goals are not perpetually-suspended after report of exception, because the arguments of the maximal goals are handed to the control process and it is possible for the control process to resume these suspended goals by instantiating some arguments. Therefore, it is possible to say that our decision, that is, reporting only the maximal goals, is theoretically correct.

```
?- a(X,Y,Z), b(Y), c(Z).

a([msg1|X],Y,Z) :- true | Y = [msg2|Y1], Z = [msg3|Z1], a(X,Y1,Z1).
b([msg2|Y]) :- true | b(Y).
c([msg3|Z]) :- true | c(Z).
```

Figure 3: Example of the Perpetually-suspended Goals

3 Suspension Mechanism and Causality Graph

3.1 Suspension Mechanism

For the KL1 implementation, we adopted a non busy-waiting manner to handle the suspensions of goals, for the efficiency[7, 8]. When the execution of a goal is suspended because of one or more uninstantiated variable(s), a goal record, containing the goal's context such as arguments of the goal and a code pointer, is hooked to the variable(s), in order to wait for the instantiation of the variable(s). And when a hooked variable is instantiated, the goal records hooked to the variable are *resumed*, and they are pushed to the *ready goal stack*, a stack of goal records ready for execution.

For instance, goals in figure 3 are represented as figure 4. In this example, goal `a/3`, `b/1`, and `c/1` are perpetually-suspended, and `a/3` is the maximal in the causality.

By observing figure 4, it can be known that the suspended goal records and their arguments represent a causality graph of these suspended goals. The goals in the downstream of the causality are accessible from the arguments of the goals in the upstream.

3.2 Maximal Goals of Perpetual Suspension

To define perpetual suspension and the maximality of causality, now we would like to introduce a notation to represent a relation between goals as follows.

- $A \succ B$ or $B \prec A$

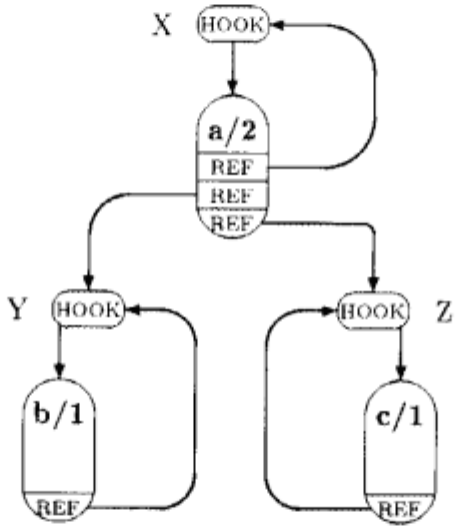

Figure 4: Goal Status

meaning: goal B is accessible from the arguments of goal A.

If there is a relation that

- $(A \succ B) \wedge (A \not\prec B)$.

goal B is in the downstream of the causality. And if there is a relation that

- $(A \succ B) \wedge (A \prec B)$.

goal A and goal B are equivalent in the causality.

Using this notation, a perpetually-suspended goal is defined as follows.

- Goal G_p is perpetually-suspended if ($\forall$ executable goal G_e, $G\_p \not\prec G_e$).

The maximal goal of perpetual suspension is defined as the goals specified by following procedure.

1. divide perpetually-suspended goals into some semi-groups, each of which consists of the goals with the equivalent causality;
2. let one goal G_i represent each semi-group to which the goal belongs;
3. G_j is maximal if ($\forall\ G_i$, $G_j \not\prec G_i$).

4 Algorithm

So far we have explained that the goal suspension mechanism generates the actual causality graph of goals during the execution. In this section, we describe the algorithm to extract the maximal goals out of causality graph of perpetual suspension, which takes advantage of the characteristics of garbage collection.

In this algorithm, we assume the usage of *copying* garbage collection scheme[1] which uses two independent semi-spaces alternatively. When one semi-space is exhausted during the execution, the active data objects in the exhausted semi-space are copied to the other semi-space, and the execution is continued on that semi-space.

The algorithm is divided into three phases:

1. detection of the occurrence;
2. search of the perpetually-suspended goals; and
3. extraction of the maximal goals.

4.1 Detection

We must consider how to detect an occurrence of perpetual suspension. It is quite difficult to do that during the normal execution. However, the perpetually-suspended goals is independent from other executable goals as we described in the previous section, and it suggests us the utilization of garbage collection. Since garbage collector traces the data objects only from the executable goals, the perpetually-suspended goals are not discovered during garbage collection. Therefore, it is only necessary to compare the number of goals before and after garbage collection, in order to detect an occurrence of perpetual suspension. To enable this, two counters for goals must be introduced. One, the *goal_counter*, is incremented or decremented on the creation or the termination of a goal respectively[2], during normal execution. The other, the *copied_goal_counter*, is used in garbage collection: during garbage collection, this counter is incremented when copying one goal from the old semi-space to the new semi-space. And when all the active data object have copied, the values of these two counters are compared, and perpetual suspension occurred if the copied_goal counter is smaller than the goal_counter.

4.2 Search for Perpetually-suspended Goals

When perpetual suspension is detected, it is necessary to search for the perpetually-suspended goal records by sweeping the old semi-space because these goal records are left un-copied. We introduced a new tag `GOAL` which only appears in goal records and enables us to distinguish goal records from other data objects during the sweeping. In addition to this, when copying one goal record from the old semi-space to the new semi-space, the `GOAL` tag of the goal record in the old semi-space must be cleared, in order to discover only the goal records left un-copied, while sweeping.

4.3 Algorithm for Discovering the Maximal Goals

In this section, we would like to describe the algorithm to extract the maximal goals from the causality graph of perpetual suspension. The algorithm is

[2] Actually, there already exists a counter for this purpose, in order to detect the termination of computation. Thus no overhead is required during the normal execution.

divided into two phases, that is, the *sweep-and-copy* phase and the *mark-and-sift* phase.

4.3.1 Sweep-and-copy Phase

In this phase, all the perpetually-suspended goals are found out and copied to the new semi-space, because the information on the perpetually-suspended goals, including argument vector, must be reported as we described in section 2.2. The procedure is as follows.

```
procedure Sweep-and-copy
begin
        while (goal_counter > copied_goal_counter) do
                begin
                        sweep old semi-space;
                        if   a goal record G is discovered then
                             begin
                                  copy G to the new semi-space;
                                  register G to the maximal goal candidate table;
                                  increment copied_goal_counter;
                                  copy all the data objects accessible from G;
                             end
                end
end
```

Maximal goal candidate table is realized as a list of goal records in the old semi-space, in the actual implementation.

During the copying of data from the old semi-space to the new semi-space, the copied_goal counter is also incremented on copying one goal record.

The statuses of the old and new semi-spaces when this phase has finished are shown in figure 5. In this figure, each G_i is a maximal goal candidate, that is, the goal is found out during memory sweep, and $CP - G_i$ is the set of data objects which were accessible from the G_i. There may exist some goal records in $CP - G_i$ from which G_i is accessible, that is, G_i may be a member of a looped causality. In this case, G_i can be regarded as a representative of the loop. G_i was found out before G_j, if $i < j$.

There is a relation between G_is that $G_i \not\succ G_j$, if $i < j$.

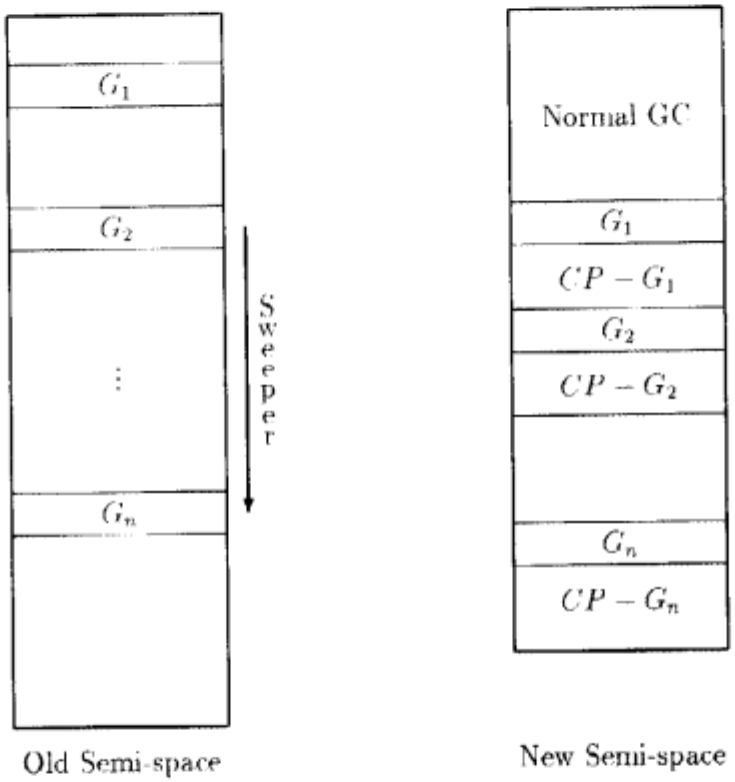

Figure 5: Old & New Semi-spaces after Sweep-and-Copy Phase

This can easily be proved as follows.

If G_j depends on G_i and $i < j$, the goal record of G_j is accessible from an argument of G_i, hence, G_j must be found out during copying started from G_i as a root. However, G_j is not found out because $i < j$, hence, G_j does not depend on G_i.

In particular, the goal G_n in figure 5 is guaranteed to be a maximal goal, because it does not depends on $G_1, \ldots, G_{n-1}$.

4.3.2 Mark-and-sift Phase

In this phase, the dependencies of the opposite direction are examined, that is, it is examined whether the relations

- $G_i \prec G_j$, $i < j$

exist or not.

The actual maximal goals are found out by sifting the maximal goal candidate table: tracing data objects from the arguments of G_j to find out G_i $(i < j)$ accessible from G_j. For the efficiency, we assume one special bit in each memory cell used as a marking bit[3], and data objects traced once are marked to avoid tracing one data object twice or more.

The algorithm is as follows.

```
procedure Mark-and-shift
begin
    while (∃G_j in candidate table) ∧ (G_j is not marked) ∧ (∀ i ≤ j, G_i is not marked) d
        begin
            mark G_j;
            while trace from arguments of G_j and mark data objects;
                begin
                    if (find maximal goal candidate G_i) ∧ (i ≠ j) then
                    begin
                        delete G_i from the candidate table;
                    end
                end
        end
```

[3] *GC bit* can be utilized for this purpose, if any.

un-mark memory cells;

end

The tracing is done recursively from one goal record as a root. We can use the old semi-space as the stack area, in this phase.

When this procedure is finished, only the actual maximal goals remain in the maximal candidate goal table, because G_i is deleted from the candidate table, if $(G_i \prec G_j \wedge i < j)$, by this procedure. Eventually, information on each maximal goal is reported to the shōen to which that goal belong, in the form described in section 2.2.

5 Discussions

5.1 Overhead

In this section, we discuss about overhead brought about by this algorithm. It can be said that overhead is small enough if there is no perpetual suspension in the system. Considerable overhead occurs when perpetual suspension is detected, however, we are convinced that it is compensated by the efficiency brought about by this algorithm, in terms of debugging.

- There is no overhead in the normal execution because no special mechanism is required which affects the normal execution.
- There is very small $O(n)$ overhead (n is a number of goals) during garbage collection, for maintaining the copied_goal_counter, if perpetual suspension do not exist.
- There is $O(n)$ overhead (n is size of memory) during garbage collection if perpetual suspension exist, because this algorithm requires the memory sweep.

5.2 Limitations

There are some limitations of this algorithm as follows.

1. Detection cannot be real-time. Sometimes it is difficult to find out the actual cause of perpetual suspension even though the maximal goals of

perpetual suspension are found out, because the actual cause, the producer process which finishes without instantiating the shared variable, does not exist in the causality graph of perpetual suspension.

This can be improved to some extent by introducing an incremental detection mechanism using MRB scheme[2].

2. The goals which are not actually maximal in the causality may be reported. This is because there is no way to know whether a reference is a read path or a write path, in our KL1 implementation. To solve this problem, we must introduce the distinction between these two paths.

3. Currently, inter-processor perpetual suspension, in which perpetually-suspended goals exist in more than one processing element, cannot be detected, because global garbage collector has not yet been implemented on the Multi-PSI. However, it is expected that this algorithm is useful enough, because, in most cases, perpetual suspension is a problem in the initial debugging stage and it seems natural that only one processing element is utilized in the case of such initial debugging.

6 Conclusions and Future Work

We have described an algorithm to detect perpetual suspension, which

1. detects the occurrence of perpetual suspension can be detected during garbage collection and;

2. reports only the maximal goals in the causality graph of perpetual suspension.

This can greatly decrease the efforts of debugging because perpetual suspension is one of the most troublesome problems when building software in the committed-choice languages like KL1. It can be said that it is quite difficult to build practical software without such an implementation support for this problem.

However, this algorithm still has a weakness that it finds out only the goals which already had been perpetually-suspended. The debugging becomes more easily if perpetual suspension can be detected when one goal

has just been perpetually-suspended. And now we are implementing such an incremental detection mechanism for perpetual suspension, which takes advantages of the MRB scheme [2]. However, this incremental detection mechanism cannot find out perpetual suspension completely because of the MRB nature, and these two, *incremental* and *batch*, detection mechanisms must coexist to complement each other, just like two garbage collection mechanisms implemented on the Multi-PSI[9].

Acknowledgments

We would like to thank Dr. T. Chikayama, who proposed an original idea of this algorithm, and to thank Mr. N. Ichiyoshi and Mr. K. Rokusawa, for valuable discussions to improve this algorithm. We would also like to thank the ICOT Director, Dr. K. Fuchi, and the chief of the fourth research laboratory, Dr. S. Uchida, for giving us the opportunity to conduct this research.

References

[1] H. G. Baker. List processing in real time on a serial computer. *Commun. ACM*, 21(4):280–249, 1978.

[2] T. Chikayama and Y. Kimura. Multiple Reference Management in Flat GHC. In *Proceedings of the Fourth International Conference on Logic Programming*, Vol. 2, pp.276–293, 1987.

[3] T. Chikayama, H. Sato and T. Miyazaki. Overview of the Parallel Inference Machine Operating System (PIMOS). In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pp.230–251, ICOT, Tokyo, 1988.

[4] K. Clark and S. Gregory. PARLOG: Parallel Programming in Logic. *ACM Trans. on Programming Languages and Systems* 8(1) pp.1–49, 1986.

[5] I. T. Foster and S. Taylor. *Strand: New Concepts in Parallel Programming.* Prentice-Hall, Englewood Cliffs, N. J. 1989.

[6] A. Goto, M. Sato, K. Nakajima, K. Taki and A. Matsumoto. Overview of the Parallel Inference Machine Architecture (PIM). In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pp.208–229, ICOT, Tokyo, 1988.

[7] N. Ichiyoshi, T. Miyazaki, and K. Taki. A distributed implementation of Flat GHC on the Multi-PSI. In *Proceedings of the Fourth International Conference on Logic Programming*, pp 257–275, 1987.

[8] Y. Kimura and T. Chikayama. An Abstract KL1 Machine and Its Instruction Set. In *Proceedings of 1987 Symposium on Logic Programming*, pp 468–477, 1987.

[9] K. Nakajima, Y. Inamura, N. Ichiyoshi, K. Rokusawa, T. Chikayama. Distributed Implementation of KL1 on the Multi-PSI/V2. In *Proceedings of the Sixth International Conference on Logic Programming*, pp.436–451, 1989.

[10] E. Shapiro. *A Subset of Concurrent Prolog.* ICOT Technical Report TR-003, 1983.

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