PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT. © ICOT, 1992

Parallel and Distributed Implementation of Concurrent Logic Programming Language KL1

Keiji HIRATA Reki YAMAMOTO Akira IMAI Hideo KAWAI
Kiyoshi HIRANO Tsuneyoshi TAKAGI Kazuo TAKI

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

Akihiko NAKASE Kazuaki ROKUSAWA

TOSHIBA Corporation OKI Electric Industry Co.,Ltd.

Abstract

This paper focuses on a parallel and distributed implementation method for a concurrent logic programming language, KL1, on a parallel inference machine, PIM. The KL1 language processor is systematically designed and implemented. First, the language specification of KL1 is deliberately analyzed and properly decomposed. As a result, the language functions are categorized into unification, inter-cluster processing, memory management, goal scheduling, meta control facilities, and an intermediate instruction set. Next, the algorithms and program modules for realizing the decomposed requirements are developed by considering the features of PIM architecture on which the algorithms work. The features of PIM architecture include a loosely-coupled network with messages possibly overtaken, and a cluster structure, i.e. a shared-memory multiprocessor portion. Lastly, the program modules are combined to construct the language processor. For each implementation issue, the design and implementation methods are discussed, with proper assumptions given.

This paper concentrates on several implementation issues that have been the subjects of intense ICOT research since 1988.

1 Introduction

In the Fifth Generation Computer Systems Project, ICOT has been, simultaneously, developing a large-scale parallel machine PIM [Goto *et al.* 1988] [Imai *et al.* 1991], designing a concurrent logic programming language KL1 [Ueda and Chikayama 1990], and investigating the efficient parallel implementation of KL1 on PIM [ICOT 1st Res. Lab. 1991]. These subjects are closely related and have been evolving together.

The KL1 language has several good features: a declarative description, simple representation of synchronization and communication, symbol manipulation, parallelism control, and portability. Similarly, PIM architecture, also, has a number of good features: high scalablity, general purpose applicability, and efficient symbolic computing.

When implementing KL1 on PIM, various difficulties appear. However, the parallel and distributed implementation of KL1 must bridge the semantic gap between PIM and KL1 so that programmers can enjoy the KL1 language as an interface for general-purpose concurrent/parallel processing [Taki 1992].

ICOT has implemented KL1 on Multi-PSI (a distributed-memory MIMD machine) and has been accumulating experience in KL1 implementation [Nakajima *et al.* 1989]. The implementation of KL1 on Multi-PSI was a preliminary experiment for our implementation.

This paper primarily focuses on a parallel and distributed implementation method for the concurrent logic programming language KL1 on a parallel inference machine PIM. Section 2 gives readers some brief background knowledge on PIM and KL1. Section 3 systematically investigates the complex connections of what part of the language specification is supported by what component(s) of the KL1 language processor. Among these components, Section 4 focuses on and discusses several key implementation issues: efficient parallel implementation within a shared-memory portion, inter-cluster processing, a parallel copying garbage collector, meta control facilities, and a KL1 compiler. Section 5 concludes this paper.

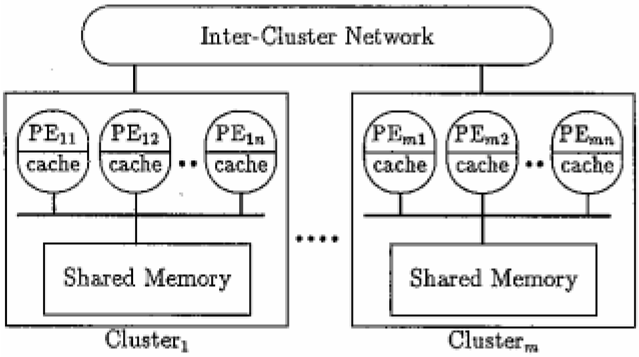


Figure 1: PIM Architecture

2 Overviews of PIM and KL1

2.1 PIM

Figure 1 shows the PIM architecture [Goto *et al.* 1988] [Imai *et al.* 1991]. PIM architecture assumptions and features are described below.

One of the features of PIM architecture is its hierarchy. Up to about ten processing elements (PEs) are interconnected by a single bus to form a structure called a "cluster" in which main memory is shared. Here, the bus can be regarded as a local network. Many clusters can be interconnected by a global network. Within a cluster, inter-PE communication can be realized by short-delay high-throughput data transfer via the bus and the shared memory. Thus, PEs within a cluster share their address spaces, and each PE has its own snooping cache. The instruction set of a PE includes `lock&read`, `write&unlock`, and `unlock` as basic memory operations.

Inter-cluster communication, though, may pass messages through some relay nodes and over long distances. Thus, inter-cluster communication increases the time delay and decreases the throughput. The address spaces of distinct clusters are separated, of course. The network delivers message packets to destinations while reading their header and tailer information.

PIM architecture assumes the following property for the inter-cluster loosely-coupled network. If PEs send and/or other PEs receive message packets, the order of packets does not obey the FIFO rule. Even in one-PE-to-one-PE communication, the FIFO rule is not obeyed. This assumption comes from the following hardware characteristics of PIM architecture. The reasons for this assumption are as follows. One is that there may be more than one path between two clusters [1]. The other is that when more than one PE within a cluster simultaneously sends message packets, it is not determined that which packet will be launched first into the network. In this sense, in the loosely-coupled network of PIM, messages are possibly *overtaken* in the network.

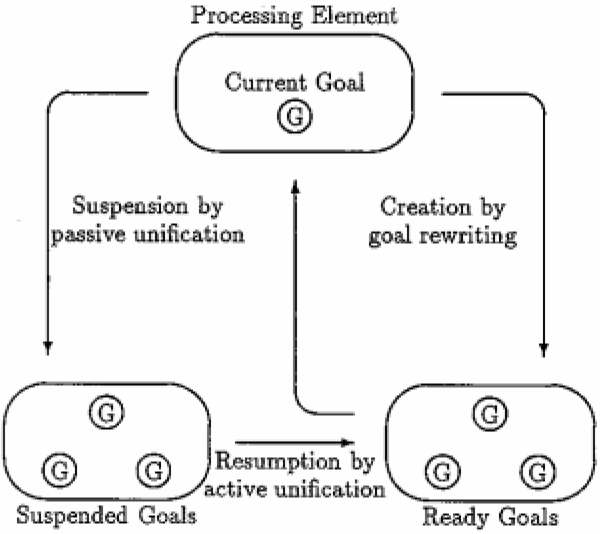

Processing Element
Current Goal
G
Suspension by passive unification
Creation by goal rewriting
G
G
G
Resumption by active unification
G
G
G
Suspended Goals
Ready Goals

Figure 2: KL1 Execution Image

2.2 KL1

KL1 is a kernel language for the PIM based on the GHC (Guarded Horn Clauses) language [Ueda and Chikayama 1990]. Figure 2 shows our KL1 execution image. A clause of a KL1 program can be viewed as a rewrite rule, which rewrites to the body goals a goal that succeeds the guard unification and satisfies the condition (guard), and has a form as follows:

$$\underbrace{p :- g_1, \ldots, g_m}_{guard\ part} \mid \underbrace{q_1, \ldots, q_n}_{body\ part}.$$

Where p, g_i, and q_i stand for predicates. This rewriting of a goal is also called *reduction*. The execution model has a goal pool which holds the goals to be rewritten. Goals are regarded as lightweight processes. Basically, guard goals $g_1, \ldots, g_m$ and body goals are reduced concurrently, thus yielding parallelism.

Goal (process) communication is realized as follows. Suppose that more than one goal shares a variable. When a goal binds a value to the shared variable, a clause for rewriting the other goal that shares the variable may be determined. The value which is instantiated to the shared variable controls the clause selection; this is the communication between KL1 goals.

Synchronization is realized as follows. When a goal is going to determine which clause can be used for rewriting, and the variables included in the goal are uninstantiated, the unification and the guard execution may be deferred since there is not enough information for the clause selection. The uninstantiated variables are supposed to be shared and the other goal is expected to bind

[1] However, the routing of the PIM network is not adaptive.

a value to the variable afterwards. Consequently, the suspended goal reduction waits for variable binding for the clause selection. That is, variable instantiation realizes data-flow synchronization. Actually, the KL1 language processor must deal efficiently with frequent suspension and resumption.

Even if more than one clause can be used for rewriting, just one clause is selected indeterminately. A vertical bar between the guard part and the body part '|', called a commit operator, designates indeterminacy. Since it is sufficient to hold a single environment for each variable, efficient implementation is expected.

One of features of the KL1 language is the provision of simple yet powerful meta control facilities as follows: goal execution control, computation resource management, and exception handling. These are essential for designing efficient parallel algorithms and enabling flexible parallel programming. Usually, operating systems perform meta-control on a process basis. However, the KL1 language aims at fine-grain parallelism, and the KL1 language processor reduces a large number of goals in parallel. Therefore, it is inefficient and impossible for a programmer or the runtime system [2] to control the execution of each goal. Consequently, KL1 introduces the concept of a shoen [3] [Chikayama *et al.* 1988]. A shoen is regarded as a goal group or a task with meta-control facilities. An initial goal is given as an argument to the built-in predicate shoen; descendant goals belonging to the shoen are controlled as a whole. Descendant goals inherit the shoen of the parent goal. Shoens are possibly nested as well; the structure connecting shoens is a tree.

Moreover, to realize sophisticated mapping of parallel computation, *priority* and *location specification* are introduced; that is, they can be used for programming speculative computation and load balancing. If a programmer attaches an annotation to a body goal e.g. p@priority(N), this tells the runtime system to execute goal p at priority N. Moreover, a goal can have a location specification e.g. p@cluster(M); this designates the runtime system to execute the goal p in the M th cluster. These two specifications are called *pragmas*. These pragmas never change the correctness of a program although they change the performance drastically.

3 Systematic Design of KL1 Language Processor

When implementing KL1 on PIM, various kinds of difficulties appear. Firstly, although the PIM architecture adopts a hierarchical configuration, the KL1 implementation has to provide a uniform view of the machine to programmers. Secondly, it is difficult to determine to what extent a runtime system should support the functions of KL1 and which functions it should support within the specification of KL1. For instance, since the KL1 language does not specify the goal-scheduling strategy, a runtime system can employ any scheduling algorithm. However, both the general-purpose and the efficient algorithm are generally difficult to develop. Thirdly, for efficient implementation, it is important to employ algorithms which include fewer bottlenecks in terms of parallel execution. Lastly, the KL1 language processor is complex and of a large scale.

Therefore, it is a promising idea to be able to overcome these difficulties by systematically designing a language processor as follows. Firstly, the given language specification must be deliberately analyzed and properly decomposed. Then, the algorithms and the program modules for realizing the decomposed requirements must be developed by considering the machine architecture on which the algorithms work. Lastly, the designer must construct the language processor by combining the program modules. A good combination of these modules will yield an efficient implementation. We designed the KL1 language processor on a loosely-coupled shared-memory multiprocessor system (PIM) by following these guidelines.

3.1 Requirements

At first, we summarize the required functions of the KL1 language processor into the four items in the leftmost column of Table 1. These items are the result of analysis and decomposition of the KL1 language specification. The KL1 language processor may look like the kernel of an operating system.

Next, mechanisms which satisfy these requirements are divided into those supported by a compiler and those supported by a runtime system. Furthermore, mechanisms by the runtime system are divided into two levels according to the machine configuration of PIM: shared-memory level and distributed-memory level (the topmost row of Table 1).

Some of the technologies used for KL1 implementation on single-processor systems may be expanded to shared-memory multiprocessor systems. That is because both systems suppose a linear memory address space. However, it may not be straightforward to expand the single-processor technologies to distributed-memory multiprocessor systems in general. Of cource, that is mainly because distributed-memory systems provide a non-linear memory address space. Thus, the techniques used for distributed-memory systems are possibly quite different from those for a single-processor system.

The contents of Table 1 show our solutions; that is,

[2] The software modules of the KL1 language processor executed at run time are called a runtime system as a whole. For instance, the runtime system may include an interpreter, firmware in microcode, and libraries. On the contrary, compilers, assemblers and optimizers are not included in a runtime system.

[3] Shoen is pronounced, '*show*' '*N*'.

Table 1: Implementation Issues of this Paper

	Compiler	Runtime System	
		Shared-memory Level	Distributed-memory Level
Unification	Decomposition	Suspension and Resumption	Message Protocol
Memory Management	Reuse inst.	Local GC	Export and Import Tables Weighted Export Count
Goal Scheduling	TRO	Automatic Load Balancing	
Meta-control			
Execution Control		Termination Detection	Foster-parent Weighted Throw Count Message Protocol
Resource Management		Resource Caching	
Exception Handling			

what techniques are used for parallel and distributed KL1 implementation. Each item in the leftmost column of the table is mentioned below.

3.1.1 Unification

Goals are distributed all over a system for load balancing and may share data (variables and ground data) for communication. Logical variables remain resident at their original location. Consequently, not only intra-cluster but also inter-cluster data-references appear. During unification, goals have to read and write the shared data consistently and independently from the timings and locations of goals and data. Thus, mechanisms for preserving data consistency are needed.

As described above, goals are rewritten in parallel and, thus, variable instantiations occur independently from each other. Suspension and resumption mechanisms based on variable bindings control goal execution and realize data-flow synchronization.

Hence, our KL1 implementation must realize the mechanisms for data consistency, synchronization, and unification in a parallel and distributed environment. Moreover, since a major portion of the CPU time is spent for unification, the algorithm should be concerned with efficiency.

3.1.2 Memory Management

Logical variables inherently have the single-assignment property. The single assignment property is very useful to programmers, but gives rise to heavy memory consumption. Since the KL1 language does not backtrack, KL1 cannot perform memory reclamation during execution as Prolog does. Thus, an efficient memory management mechanism is indispensable for the KL1 language processor. The issues associated with memory management are allocation, reclamation, working-set size, and garbage collection. To achieve high efficiency, not only must the algorithms and the data structure of the runtime system be improved, but also a compiler has to generate effective codes by predicting the dynamic behavior of a user program as much as possible.

3.1.3 Goal Scheduling

The KL1 language defines goal execution as concurrent. Thus, the system is responsible for the exploitation of actual parallelism. One implementation issue associated with goal scheduling is determining which goal scheduling strategies have high data locality, yet keep the number of idle PEs to a minimum.

Further, the KL1 language provides the concept of goal priority; each KL1 goal has its own priority as explicitly designated by a programmer. Then, goals with higher priorities are *likely* to be reduced first. Goal prioritization in KL1 is weak in some respect. Under the goal priority restriction, it is crucial to achieve load balancing.

3.1.4 Meta Control Facilities

The goals of a shoen may actually be distributed over any clusters, and, thus, goals may be reduced on any PE in the system. Since the system operates in parallel, shoens are loosely managed; it is simply guaranteed that each operation will finish eventually. That is, it is impossible to execute a command simultaneously to all the goals of a shoen.

A shoen has two streams as arguments of the shoen built-in predicate; one is for controlling shoen execution, and the other is for reporting the information inside the shoen. A shoen communicates with outside KL1 processes through these two streams. Messages, such as `start`, `stop`, and `add_resource`, enter the control stream from the outside. Messages, such as `terminated`, `resource_low`, and `exception` return to the report stream from the inside.

It is very difficult to evaluate the CPU time and memory space spent for computation when goals are distributed and executed in parallel. Therefore, the current system regards the number of reductions as a measure of the computing resources consumed within the shoen.

The exceptions reported from a shoen include illegal input data, unification failure[4], and perpetual suspension.

Some examples of shoen functions are shown below.

`stop` message: When a `stop` message is issued in the control stream of a shoen, the system has to check whether or not the goals to be reduced belong to the shoen, and, if they do, the shoen changes its status to `stop` as soon as possible. The `stop` message is propagated to the nested descendant shoens.

Resource Observation: The system always watches the consumption of computation resources, that is, the total number of times goals belonging to each shoen are reduced over the entire system. If the amount of consumption within a shoen is going to exceed the initial amount of supplied resources, the system stops the reduction of shoen goals and, then, issues the `resource_low` message on the report stream, viz. a supply request for a new resource.

Exception Handling: When a programmer or the system creates an exception during the reduction of a goal in a shoen, the shoen responsible recognizes the exception and converts the exception information to a report stream message [5]. The exception of the KL1 language is concerned with illegal arguments, arithmetic, failure, perpetual suspension and debugging. An exception message on the report stream indicates which goal caused what exception and where. Additionally, the exception message includes variables for a continuation given from the outside; the other process can designate a substitute goal to be executed, instead of the goal causing the exception.

3.2 Overview of Implementation Techniques

ICOT developed the Multi-PSI system in 1988 [Nakajima *et al.* 1989]. The KL1 system is running on the Multi-PSI. The architecture of PIM is very different from that of Multi-PSI in the following two points. One is that PIM has a loosely-coupled network with messages possibly overtaken. The other is that PIM has cluster structures that are shared-memory multiprocessors. Due to these features, PIM attains high performance, and, at the same time, the complexity of the KL1 language processor increases.

This section describes many of the implementation techniques we have been developing for such an architecture. Among these techniques, the issues which this paper focuses on are listed in Table 1.

3.2.1 Unification

The synchronization and communication of KL1 are realized by read/write operations to variables and suspension/resumption of goal reduction during unification. These operations are described below.

Passive Unification and Suspension: Passive unification is unification issued in the guard part of KL1 programs. The KL1 language does not allow instantiation of variables in its guard part. The guard part unification is nonatomic. Since KL1 is a single-assignment language, once a variable is instantiated, the value never changes. This means that passive unification is simply the reading and comparing of two values. From the implementational point of view, basically only read operations to variables are performed. Thus, no mutual exclusion is needed in the guard part.

If goal reduction during the guard part is suspended, the goal is hooked to variables. Here, we have an assumption that almost all goals wait for a single variable to be instantiated afterwards. Therefore, an optimization may be taken into account; the operation for the goal suspension is just to link the goal to the original variable. If multiple uninstantiated variables suspend goal reduction, however, the goal is linked to the variables through a special structure for multiple suspension. During passive unification, only these suspension operations modify variables; the operations are realized by the compare & swap primitive.

Active Unification and Resumption: Active unification is unification issued in the body part of KL1 programs. The KL1 variables are allowed to be instantiated only in the body part. When an instantiation of a shared variable occurs, if goals are already hooked to the variable, these goals have to be resumed as well as the value assignment. When instantiating a variable, since other PEs might be instantiating the variable simultaneously, mutual exclusion is required. We also adopt compare & swap as the mutual exclusion primitive.

When unifying two variables, one variable has to be linked to another to make the two variables identical. At this time, other PEs might be unifying the same two variables. Therefore, imprudent unification operation might turn out to generate a loop structure and/or dangling references. To avoid these, the following linking rule should be obeyed: the variable with the lowest address is linked to the one with the highest.

Section 4.1 describes the implementation of unification in detail.

[4] Notice that the unification failure of a KL1 goal does not influence the outside of a shoen. In this sense, the reduction of a KL1 goal never fails, unlike GHC.

[5] The mechanism for creating and recognizing exceptions is similar to catch-and-throw in LISP.

3.2.2 Inter-cluster Processing

In a KL1 multi-cluster system, more than one PE in each cluster reduces goals in parallel. If a goal reduction succeeds, there are two kinds of new goal destination: the cluster that the parent goal belongs to and the other cluster. If the other cluster is designated for load balancing, the runtime system *throws* the new goals to the clusters. If the arguments of a goal to be thrown are references to variables and structures, the references across clusters consequently appear, these are called *external references.* Here, suppose that a new goal with reference to data in cluster A is thrown to cluster B. Then, original cluster A *exports* the reference to the data to cluster B, and foreign cluster B *imports* the reference to the data from cluster A. Exportation and importation are also implemented by message sending. Multiple reference across clusters inevitably occurs.

An external reference is straightforwardly represented by using the pair <*cl, addr*> where *cl* is the cluster number in which the exported data resides, and *addr* is the memory address of the exported data. This representation of an external reference provides programmers with a linear memory space.

However, this implementation causes a crucial problem; efficient local garbage collection is impossible. Here, *local* means that garbage collection is performed locally within a cluster. See Section 4.3 for more details on garbage collection. Since our local garbage collector adopts a stop and copy algorithm (Section 4.3), the locations of data move after garbage collection. At that time, all of the new addresses of moved data should be announced to all other clusters. Thus, straightforward representation would make cluster-local garbage collection very inefficient.

Section 4.2 shows our solution to this problem and discusses more detailed inter-cluster processing subjects.

3.2.3 Memory Management

As described in Section 3.1.2, the implementation of memory management should pay close attention to allocation, reclamation, working set size, and garbage collection.

Allocation and Reclamation: A cluster has a set of free lists for pages and supports any number of contiguous pages [6]. These are called global free lists. The size of pages is uniform; supposedly the integral power of two [7]. A PE has a set of free lists for data objects, the sizes of which are less than the page size. These are called private free lists. Actual object size is rounded up to the closest integral power of two; the private free lists just support the quantum sizes of 2^n. Moreover, objects contained in a page are uniform in size.

A PE allocates an object as follows. When a PE requires an object which is smaller than a page, the PE first tries to take an object from an appropriate private free list. If a PE runs out of a private free list and fails to take an object, then the PE tries to take a new page from the global free lists. If it succeeds, the PE partitions the page area into objects of the size the PE requires, recovers the starved free list and, then, uses an object. Otherwise, if a PE cannot take a proper page area from a global free list, the PE tries to extend the heap to allocate a new page area on demand. When a PE requires an object which is larger than a page, the PE tries to take new contiguous pages from global free lists. Otherwise, the PE tries to extend a heap to allocate new contiguous pages as above.

When a PE reclaims a large or small object, it is linked to the proper free list.

The features of this scheme are as follows:

- Since a PE has its own private free lists for small objects, the access contention to global free lists and the heap is alleviated.
- A PE usually just links garbage objects to and takes new objects from appropriate free lists; it leads the small runtime overhead for allocation and reclamation [8].
- Since every PE handles its private free lists using push and pop operations (obeying the LIFO rule), the working set size can be kept small.
- Since the size of small objects is rounded up to the nearest 2^n, the number of private free lists to be managed decreases, and the deviation of private free list lengths can be alleviated to some extent. Additionally, the fragmentation within a page is prevented, though some objects might contain unused areas.
- Since this scheme does not join two contiguous objects, unlike the buddy system, its runtime overhead of reclamation is kept small.

On the other hand, when the free list of some size run out, our KL1 language processor does not partition a large object into smaller ones, but allocates a new page. This is mainly because, due to too much partitioning, it is likely that garbage collection will be invoked even if only slightly large object is required. The other reasons are as follows. In general, it is inefficient to incrementally partition a small object into even smaller objects. The overhead for searching an object to be partitioned is needed. Also, in our KL1 language processor, a local stop-and-copy garbage collector (described just below, (2)) collects garbages and rearranges the heap area efficiently.

[6] Currently, there are 15 kinds of free lists for supported pages: $1 \sim 15-and-more$.

[7] The size of a page is currently 256 words.

[8] A module of PIM, PIM/p, has dedicated machine instructions for handling free lists, *push* and *pop*.

Furthermore, a KL1 compiler optimizes memory management by generating codes not only for allocation and reclamation but also to reuse data structures utilizing the MRB scheme [Chikayama and Kimura 1987] (Section 4.6.4).

Garbage Collection: Our KL1 language processor performs three kinds of garbage collections

(1) local real-time garbage collection using the MRB scheme
(2) local stop-and-copy garbage collector
(3) real-time garbage collection of distributed data structures across clusters.

Since (1) can reclaim almost any garbage object, (2) is needed, eventually. (1) has a very small overhead and can defer the invocation of (2). Moreover, in a shared-memory multiprocessor, it is important that (1) does not destroy data on snooping caches and keeps the working set size of an application program small [Nishida *et al.* 1990], unlike (2). Section 4.3 discusses the parallel copying garbage collector (2) in detail. Section 4.2.2 discusses our method for reclaiming data structures referred to by external reference (3) in detail.

3.2.4 Goal Scheduling

The aim of goal scheduling is to finish the execution of application programs earlier. It is impossible for a programmer to schedule all goals strictly during execution. In particular, in the knowledge processing field, there are many programs in which the dynamic behavior is difficult to predict. The optimum goal scheduling depends on applications, and, thus, there are no general-purpose goal scheduling algorithms. Hence, a programmer cannot avoid leaving part of the goal scheduling to a runtime system. Then, PEs within a cluster share their address spaces, and the communication between them is realized with a relatively low overhead. Optimistically thinking, the performance will pay for the overhead of the automated goal-scheduling within a cluster as the number of PEs increases. However, when the automated goal-scheduling for inter-cluster does not work well, the penalty is even greater. Consequently, the KL1 language processor adopts automated goal-scheduling performed within a cluster and manual goal-scheduling among clusters.

Furthermore, the runtime system should schedule goals fairly by managing priorities. Section 4.4 discusses the implementation of goal scheduling.

3.2.5 Meta Control Facilities

The meta control facilities of KL1 are provided by a shoen. The implementation model for a shoen on a distributed environment introduces a *foster-parent* to prevent bottlenecks and to realize less communication. A foster-parent is a kind of proxy shoen or a branch of a shoen; the foster-parents of a shoen are located on clusters where the goals of the shoen are reduced.

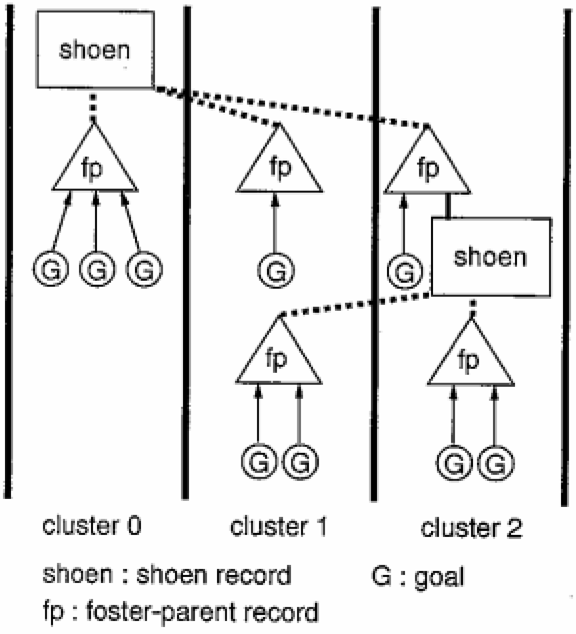


Figure 3: Relationship of Shoen and Foster-parents

A shoen and a foster-parent are realized by record structures which store their details, such as status, resources, and number of goals. Figure 3 shows the relationship between shoens, foster-parents and goals.

As in Figure 3, a shoen controls its goals and the descendant shoens resident in a cluster through a foster-parent of the cluster. A shoen directly manages its foster-parents only. Then, a foster-parent manages the descendant shoens and goals.

A shoen is created by the invocation of the shoen predicate. At that time, a shoen record is allocated in the cluster to which the PE executing the shoen predicate belongs. Next, when a goal arrives at a cluster but the foster-parent of its shoen does not yet exist, a foster-parent is created for the goal execution automatically. During execution, new goals and new descendant shoens are repeatedly created and terminated. When all goals and descendant shoens belonging to a foster-parent are terminated, the foster-parent is terminated, too. Further, when all foster-parents belonging to a shoen are terminated, the shoen is terminated.

On comparing a shoen record and a foster-parent record of our implementation with those of the Multi-PSI system, ours must hold more information because of the PIM network with messages possibly overtaken. That is, in our KL1 system, the automatons to control a shoen and a foster-parent require more transition states.

Consequently, in terms of implementing a shoen and a foster-parent, we have to pay special attention to efficient protocols between a shoen and its foster-parents which work on the loosely-coupled network of PIM (messages are possibly overtaken in the PIM). Another point requiring attention is that, since parallel accessing might become a bottleneck, the system should be designed so that such data do not appear, i.e. less access contention. Section 4.5 describes the parallel implementation of a shoen and a foster-parent in more detail.

3.2.6 Intermediate Instruction Set

As described so far, the KL1 language processor is too large and complex to be implemented directly in hardware or firmware. To overcome this problem, we adopted a method suggested by Prolog's Warren Abstract Machine (WAM) [Warren 1983] where the functions of the KL1 language processor are performed via an intermediate language, KL1-B. The advantages of introduction of an intermediate language include: code optimization, ease of system design and modification, and high portabilty.

The optimization achieved at the WAM level brings about more benefits than the peep-hole optimization since the intermediate instruction sequence reflects the meanings of the source Prolog program. Similarly, the optimization at the KL1-B level gains more than the peep-hole optimization. Details on the optimization are described in Sections 4.6.4 and 4.6.5.

If the specification of the KL1-B instruction set is fixed, it is possible to independently develop a compiler for compiling KL1 into KL1-B and a runtime system executing the KL1-B instructions. If a runtime system can be designed so that it absorbs the differences in hardware architecture, the machine-dependent parts of the KL1 language processor are made clear, and portability is improved.

3.2.7 Built-in Predicates

This section mentions the optimization techniques on the implementation of the built-in predicates `merge` and `set_vector_element`. These techniques were originally invented for the Multi-PSI system. Our KL1 language processor basically inherits the techniques.

merge: The `merger` predicate merges more than one stream into another. It is useful for representing indeterminacy; actually, the `merge` predicate is invoked frequently in practical KL1 programs, such as the PIMOS operating system [Chikayama *et al.* 1988]. Although a program for a stream merger can be written in KL1, the delay is large. Thus, it is profitable to implement the merger function with a constant delay by introducing the `merge` built-in predicate.

Let us consider a part of a KL1 program:

```
.., p(X), q(Y), merge(X,Y,Z), ..
```

When predicate `p` is to unify `X` and its output value, a system merger is invoked automatically within the unifier of `X`. The same thing happens as `Y` of `q`. See [Inamura *et al.* 1988] for a more detailed discussion.

set_vector_element: To write efficient algorithms without disturbing the single-assignment property of logical variables, the primitive can be used as follows in the KL1 language:

```
set_vector_element(Vect, Index, Elem,
                         NewElem, NewVect)
```

When an array `Vect`, its index value `Index`, and a new element value `NewElem` are given, this predicate binds `Elem` to the value at the position of `Index` and `NewVect` to a new array which is the same as `Vect` except that the element at `Index` is substituted for `NewElem`. Using the MRB scheme, our KL1 language processor detects a situation that `NewVect` is obtained in constant time. That is, the situation is that the reference to `Vect` is single, and, thus, destructive updating of the array is allowed. See [Inamura *et al.* 1988] for a more detailed discussion.

4 Implementation Issues

This section focuses on several important implementation issues which ICOT has been working on intensively for the past four years.

Our implementation mainly takes the following into account:

- *Smaller and shorter mutual exclusion within a cluster*
  If the locking operation is effective over a wide area or for a long time, system performance is seriously degraded due to serialization. To avoid this, scattered and distributed data structures are designed, and only the *compare & swap* operation is adopted as a low-level primitive for light mutual exclusion [9].
- *Less communication; i.e., fewer messages*
  Since inter-cluster communication costs more than inner-cluster communication, mechanism for eliminating redundant messages are effective.
- *Main path optimized while enduring low efficiency in rare cases*
  Since the efficiency of rare cases does not affect total performance, the implementation for handling the rare cases is simplified and low efficiency is endured. This is important for reducing code size.

Important hardware restrictions to be taken into account are:

[9] Higher-level software locks contain this primitive.

- *Snooping caches within a cluster; data locality has a great effect*
  It is important to keep the working set of each PE size small. This leads to a reduction in the shared bus traffic and increase in the hit ratio of the snooping caches.
- *Messages are possibly overtaken in the loosely-coupled network of PIM*
  The number of shoen states and foster-parent states to be maintained increases. The message protocol between clusters should be carefully designed.

4.1 Unification

The unification of variables shared by goals realizes synchronization and communication among goals. Since more than one PE within a cluster performs unification in parallel, mutual exclusion is required when writing a value to a variable.

Since unification is a basic operation of the KL1 system, efficiency greatly affects total performance. At first, this section shows simple and efficient implementation methods of unification. Next, since problems associated with the loosely-coupled network of PIM occur, a distributed unification algorithm which works consistently and efficiently on the network is presented.

4.1.1 Simplification Methods

There are two ways to simplify the unification algorithm as follows.

Structure Decomposition: A KL1 compiler decomposes the unification of a clause head. For example, (a) of the following program is decomposed to (b) at compile time.

```
p([f(X)|L]) :- true              | q(X), p(L).   (a)
p(A) :- A = [Y|L], Y = f(X)  | q(X), p(L).   (b)
```

Thus, the compiler can generate more efficient KL1-B code corresponding to (b).

Substitution for System Goals: In rare cases, a runtime system automatically substitutes part of the unification process with special KL1 goals. This can alleviate the complexity of a unification algorithm; implementors need not pay attention to mutual exclusion of the part. For example, let us consider the following two rare cases.

- A compare & swap failure (another PE has modified the value); If this happens, then the following KL1 goal is automatically created and scheduled as if it were defined by a user:

  ```
  unify_retry(X,Y) :- true | X = Y.
  ```

  The above X and Y are unified to variables one at least of which has failed compare & swap during unification.

- Active unification of two structures is invoked; All elements of the two structures should be unified, however, the operation is rather complex (the ordinal implementation uses stacks like Prolog). To simplify the operation for rare cases, a special KL1 goal is ordinarily created and scheduled. For example, if two active unification arguments are both lists, the following goal is created.

```
list_unifier([X1|X2], [Y1|Y2]) :- true |
        X1 = Y1, X2 = Y2.
```

4.1.2 Distributed Implementation Based on Message Passing

The principle of the protocol for distributed unification is as follows. A read/write operation to an external reference cell (Section 4.2.1) basically causes a corresponding request message to be launched to the network. However, redundant messages are eliminated as much as possible.

Distributed Passive Unification: Passive unification has two phases: reading and comparing. First, to execute the read operation on an external reference cell is to send a read message to the foreign exported data. If the exported data has become a ground term (an instantiated variable), an answer_value message returns. If the exported data is still a variable, the request message is hooked to the variable. If the data is an external reference cell, the read message is forwarded to the cluster to which the cell refers.

Next, the answer_value message arrives at the original cluster. Then, the returned value is assigned to the external reference cell, and the goal waiting for the reply message is resumed. Eventually, the goal reduction is going to compare the two values. Moreover, the import table entry for the cell can be released.

The efficient implementation of inter-cluster message passing itself is presented in Section 4.2.

Safe and Unsafe Attributes: If an argument of active unification is an external reference cell, the active unification has to realize the assignment in a remote cluster. Sending a unify message to the exported data assigns a value to the original exported data. However, in general, the unification of two variables from distinct clusters may generate a reference loop across clusters. In order to avoid creating such reference loop, we introduce the concept of *safe/unsafe external references* [Ichiyoshi *et al.* 1988]. When there is active unification between a variable and an external reference cell, and the external reference cell is *safe*, it is possible that the variable is bound to the external reference cell. If the external reference cell is *unsafe*, a unify message is sent to the exported data.

4.2 Inter-cluster processing

4.2.1 Export and Import Tables

Export Table: As described in Section 3.2.2 , straightforward implementation of an external reference makes cluster-local garbage collection very inefficient.

In order to overcome this problem, each cluster introduces an *export table* to register all locations of data which are referenced from other clusters (Figure 4). That is, exported data should be accessed indirectly via the export table. Thus, the external reference is represented by the pair <*cl, ent*>, called the *external reference ID*, where *ent* is the entry number of the export table. As the export table is located in the area which is not moved by local garbage collection, the external reference ID is not affected by local garbage collection. Changes in the location of exported data modify only the contents of export table entries.

Since exported data is identified by its external reference ID, distinct external reference IDs are regarded as distinct data even if they are identical. To eliminate redundant inter-cluster messages, exported data should not have more than one external reference ID. Thus, every time a system exports an external reference ID, the system has to check whether or not the external reference ID is already registered on the export table.

Import Table: In order to decrease inter-cluster traffic, the same exported data should be accessed as few times as possible. Hence, each cluster maintains an *import table* to register all imported external reference IDs. The same external references in a cluster are gathered into the same internal references of an *external reference cell* (EX in Figure 4).

Then, exported data is accessed indirectly via the external reference cell, the import table, and the export table.

The external reference cell is introduced so that it can be regarded equally as a variable. Operations to a variable are substituted for the operations to the external reference cell.

Every time the system imports an external reference ID, the system has to check whether or not the external reference ID is already registered in the import table. Thus, the import table entry and the external reference cell point to each other.

4.2.2 Reclamation of Table Entries

As described above, the export table is located in the area which is not moved by local garbage collection.

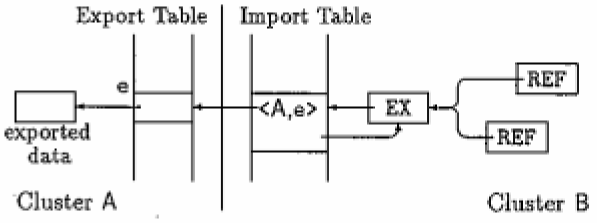


Figure 4: Export and Import Tables

During local garbage collection, data referred to by an export table entry should be regarded as active data, because it is difficult to know whether or not the export table entry is referred to by other clusters immediately. Therefore, without an efficient garbage collection scheme for the export table, many copies of non-active data would survive, these reducing the effective heap space and decreasing garbage collection performance.

One way of managing table entries efficiently is for table entries to be reclaimed incrementally. Below, we describe a method for reclaming table entries in detail.

Let us consider utilizing local garbage collection. Execution of local garbage collection might release the external reference cells. This leads to the release of import table entries and the issue of `release` messages to the corresponding export table entries. When the export table entry is no longer accessed, the entry is released. However, the reference count scheme cannot be used to manage the export table entries. This is because the increase and decrease messages for the reference counters of the export table entries are transferred through a network. Then, the arrival order of the two messages issued by the two distinct clusters is not determined in the PIM global network. This destroys the consistency of reference counters. Additionally, in the PIM network, messages are possibly overtaken. Although the reference count scheme has been improved and now requires the acknowledgment of each increase and decrease message, this will increase the network traffic.

A more efficient scheme, the *weighted export counting* (WEC) scheme has been invented [Ichiyoshi *et al.* 1988]. This is an extension of the weighted reference counting scheme [Watson and Watson 1987] [Bevan 1989] in the sense that the messages being transmitted in the loosely-coupled network also have weights. With the WEC scheme, every export table entry E holds the following invariant relation (Figure 5):

$$Weight\ of\ E = \sum_{x\ \in\ references\ to\ E} Weight\ of\ x$$

A weight is an integer. When a new export table entry is allocated, the same weight is assigned to both the export table entry and the external reference. When an import table entry is released, its weight is returned to the corresponding export table entry by the `release` message. The weight of the export table entry is decreased by the returned weight. The export table entry is detected as no longer being accessed when the weight of the entry becomes zero. Then, the entry is released from the export table. See [Ichiyoshi *et al.* 1988] for more details on the operation of the WEC scheme.

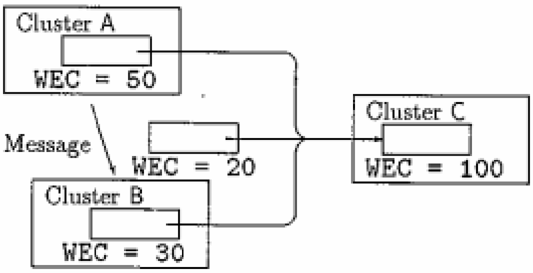


Figure 5: WEC Invariant Relation

It is important that the WEC scheme is not affected by the order in which messages arrive, and there is no need to give acknowledgment. Furthermore, the WEC scheme alleviates the cost of splitting external references.

4.2.3 Supply of Weighted Export Count

In terms of the WEC scheme, the problem of how to manage WEC when the weight of the import table entry cannot be split (when the weight reaches 1) remains.

In order to overcome this problem, we developed a *WEC supply mechanism* which is an application of the bind hook technique. The bind hook technique suspends and resumes the KL1 language (Section 2.2) [Goto *et al.* 1988].

The WEC supply mechanism works as shown in Figure 6 and 7. The current situation is that the weight of an import table entry in Cluster B reaches 1, and a goal in Cluster B issues an access command to the data in Cluster A. In this case, the message related to the access command cannot be sent, because the weight to be put on the message command cannot be got from the import table entry.

In the WEC supply mechanism, the left WEC (the weight is 1), first, is taken from the import table entry, and the import table entry is reclaimed. After that, in Cluster B, an export table entry for the external reference cell is allocated. This new external reference ID is supposed to be the return address for the reply to the following WEC supply request. At that time, the goal is hooked to the external reference cell. Eventually, Cluster B sends the `RequestWEC` message to request a new weight to Cluster A. Of course, the weight taken from the import table entry described above is returned to the corresponding export table entry by this message. Figure 6 shows the situation at that time.

When Cluster A receives the `RequestWEC` message, Cluster A adds a weight, say *W*, to the corresponding export table entry and returns the `SupplyWEC` message to Cluster B. The `SupplyWEC` message tells Cluster B to add the weight *W* to a new import table entry. In Cluster B, the suspended goal is resumed when the new import table entry is allocated. Then, the export table entry for the return address is reclaimed. Figure 7 shows the situation at that time.

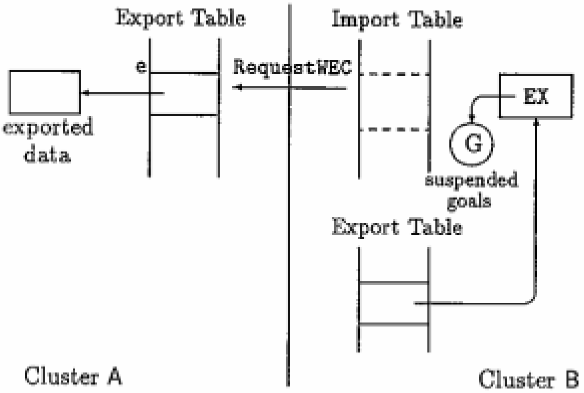


Figure 6: WEC Request Phase

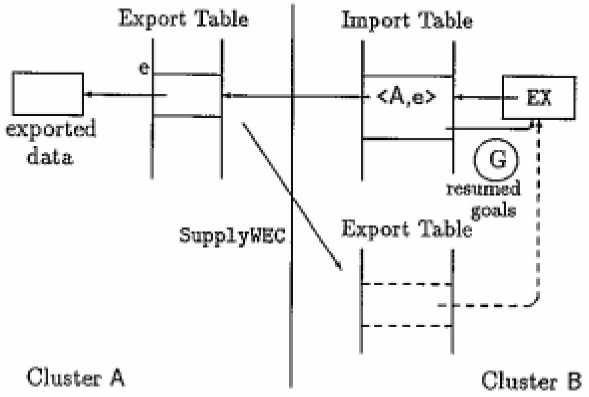


Figure 7: WEC Supply Phase

This mechanism allows the originated goal to be hooked and resumed inexpensively without additional data structures.

The KL1 language processor on Multi-PSI copes with this situation using *indirect exportation* and *zero WEC message* [Ichiyoshi *et al.* 1988]. However, the zero WEC message is a technique which is applicable to a FIFO network. As described earlier, the PIM network does not obey the FIFO rule, so the zero WEC message cannot be used in PIM. Therefore, PIM uses indirect exportation and WEC supply mechanism.

4.2.4 Mutual Exclusion of Table Entries

In order to check whether or not an external reference is already registered on the export table, a hash table is used. When an export table entry is allocated, it is registered in the hash table. When a cluster receives a release message, a PE in the cluster decreases the weight of the corresponding export table entry. If the weight reaches zero, the export table entry is removed from the hash table. Figure 8 shows the data structure of the export table and its hash table. Its hash key is the address of exported datum.

Since up to about ten PEs within a cluster share these structures and access them in parallel, efficient mutual exclusion should be realized.

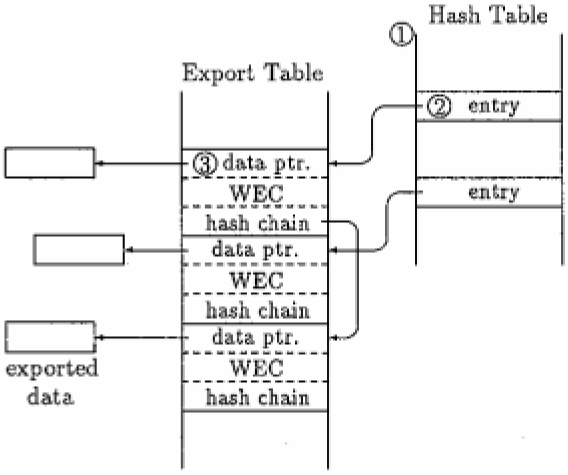


Figure 8: Data Structures of Export Table

Here, let us consider how to realize efficient mutual exclusion in the following two cases, which are typical cases of release message processing.

Case 1: A PE decreases the weight of an export table entry and the weight does not reach zero. In this case, only an export table entry is directly accessed. The export table entry should be locked, when manipulating its weight. The corresponding hash table entry does not need to be locked, because the hash chain does not change.

Case 2: A PE decreases the weight of export table entry and the weight reaches zero. In this case, the export table entry is released from hash table entry. Therefore, the export table entry should be locked for the same reason as in Case 1. The hash table entry should also be locked, when the export table entry is released from the hash chain, because other PEs may access the same hash chain simultaneously.

The problem is how to lock these structures efficiently. Here, we implemented the following three methods and evaluated their efficiency.

Method 1: Locking entire hash table and export table
Whenever a PE accesses the export table, the export table and the hash table are entirely locked. In Figure 8, location ① is locked.
Since the implementation of this method is simple, the total execution time is short. However, this method occupies a large locking region for a long time. Thus, access contention occurs very frequently.

Method 2: Locking one hash table entry
When a PE decreases the weight of an export table entry, the corresponding hash table entry (② in Figure 8) is locked.
In this method, the data structure to be locked is obviously smaller than in Method 1. However, this method has an overhead for computing the hash value of exported data even when the hash chain is not modified.

Method 3: Locking one hash table entry and one export table entry
When a PE decreases the weight of an export table entry, the export table entry (③ in Figure 8) is locked. If the weight becomes zero, the corresponding hash table entry (② in Figure 8) is locked. Then, the export table entry is released from the hash chain.
In this method, the locking of data structures is at a minimum and the frequency of access contention is low. However, implementation of this method is complicated.

In the above two cases, the static execution steps of the three methods are measured, using a parallel KL1 emulator on a Sequent Symmetry. Tables 2 and 3 show the results. In the tables, Total represents the total execution steps spent on receiving a release message. Locking region represents locking intervals, that is, how long each structure is locked.

Table 2: Locking Intervals(static steps) Case 1

	Total	Locking region		
		①	②	③
Method 1	30	23	—	—
Method 2	37	—	23	—
Method 3	32	—	0	26

Table 3: Locking Intervals(static steps) Case 2

	Total	Locking region		
		①	②	③
Method 1	61	54	—	—
Method 2	61	—	47	—
Method 3	73	—	32	27

Before evaluation, we thought that Method 1 took fewer steps than the other methods. However, there is

,actually, no great difference in the total number of execution steps. This is because the essential part of accessing the export table is complicated, and dominates the steps. In Method 1, as the ratio of the locking region to the total is relatively high, access contention to the hash table is supposed by frequent. Hence, we do not adopt Method 1.

[Takagi and Nakase 1991] tells us that WEC is effectively divided in actual programs. From this result, we assume that there are many `release` messages which just decrease the weight of WEC. That is, Case 1 occurs much more frequently than Case 2. Thus, we mostly deal with Case 1. The total execution steps of Methods 2 and 3 (37 steps and 32 steps) are almost the same, The locking intervals of Methods 2 and 3 (23 steps and 26 steps) are almost the same. It is preferable that the data structure to be locked is small. According to this discussion, we adopt Method 3 as the mutual exclusion method for the export table.

For the import table, a similar technique is used to reclaim the import table entries.

4.3 Parallel Copying Garbage Collector

Efficient garbage collection (GC) methods are especially crucial for the KL1 language processor on multiprocessor systems. Since the KL1 execution dynamically consumes data structures, GC is necessary for reclaiming storage during computation. Moreover, GC should be executed at each cluster independently since it is very expensive to synchronize all clusters.

As we described briefly in Section 3, an incremental GC method based on the MRB scheme was already proposed and implemented on Multi-PSI [Inamura *et al.* 1988], however since it cannot reclaim all garbage objects, it is still important to implement an efficient GC to supplement MRB GC.

We invented a new parallel execution scheme of stop and copy garbage collector, based on Baker's sequential stop-and-copy algorithm[Baker 1978] for shared memory multiprocessors. The algorithm allocates two heaps although only one heap is actively used during program execution. When one heap is exhausted, all of its active data objects are copied to the other heap during GC. Thus, since Baker's algorithm accesses active objects this algorithm is simple and efficient.

Innovative ideas in our algorithm are the methods which reduce access contention and distribute work among PEs during cooperative GC. Also no inter-cluster synchronization is needed since we use the export table described in Section 4.2. A more detailed algorithm is described in [Imai and Tick 1991].

4.3.1 Parallel Algorithm

Parallelization: There is potential parallelism inherent in the copying and scanning actions of Baker's algorithm, i.e., accessing S and B. Here pointer S represents the *scanning point* and B points to the *bottom* of the new heap. A naive method of exploiting this parallelism is to allow multiple PEs to scan successive cells at S, and copy them into B. Such a scheme is bottlenecked by the PEs vying to atomically read and increment S by one cell and atomically write B by many cells. Such a contention is unacceptable.

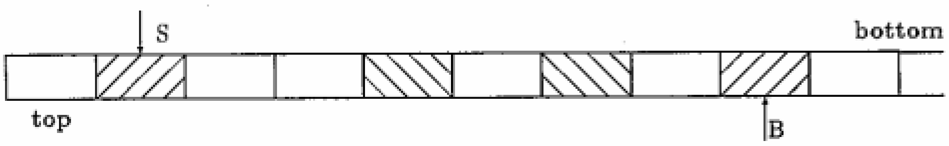
Private Heap: One way to alleviate this bottleneck is to create multiple heaps corresponding to multiple PEs. This is the structure used in both Concert Multilisp[Halstead 1985] and JAM Parlog[Crammond 1988] garbage collectors. Consider a model where each PE(i) is allocated private sections of the new heap, managed with private S_i and B_i pointers. Copying from the old space could proceed in parallel with each PE copying into its private new sections. As long as the *mark* operation in the old space is atomic, there will be no erroneous duplication of cells. Managing private heaps during copying, however, presents some significant design problems:

- Allocating multiple heaps within the fixed space causes fragmentation.
- It is difficult to distribute the work among the PEs throughout the GC.

To efficiently allocate the heaps, each PE extends its heap incrementally in *chunks*. A chunk is defined as a unit of contiguous space, that is a constant number of HEU cells (HEU ≡ Heap Extension Unit). We first consider a simple model, wherein each PE operates on a single heap, managed by a single pair of S and B pointers. The B_{global} pointer is a state variable pointing to the global bottom of the new allocated space shared by all PEs. Allocation of new chunks is always performed at B_{global}.

Global Pool for Discontiguous Areas: When a chunk has been filled, the B pointer reaches the top of the next chunk (possibly not its own!). At this point a new chunk must be allocated to allow copying to continue. There are two cases where B overflows: either it overflows from the same chunk as S, or it overflows from a discontiguous chunk. In both cases, a new chunk is allocated. In the former case, nothing more needs to be done because S points into B's previous chunk, permitting its full scan. However, in the latter case, B's previous chunk will be lost if it is separated from S's by extraneous chunks (of other PEs, for instance).

The problem of how to 'link' the discontiguous areas, to allow S to freely scan the heap, is solved in the following manner. In fact, the discontiguous areas are not



The shaded portions of the heap are owned by a PE(i) which manages S and B. Other portions are owned by any PE(j) where $j \neq i$. The two chunks shaded as '/' are referenced by PE(i) via S and B. The other chunks belonging to PE(i), shaded as '\', are *not* referenced. To avoid losing these chunks , they are registered in the global pool.

Figure 9: Chunk Management in Simple Heap Model

linked at all. When a new chunk is allocated, the B's previous chunk is simply added to a *global pool*. This pool holds chunks for load distribution, to balance the garbage collection among the PEs. Unscanned chunks in the pool are scanned by idle PEs which resume work (see Figure 9).

Uniform Objects in Size: We now extend the previous simple model into a more sophisticated scheme that reduces the fragmentation caused by dividing the heap into chunks of uniform size. Imprudent packing of objects which come in various sizes into chunks might cause fragmentation, leaving useless area in the bottom of chunks. To avoid this problem, each object is allocated the closest quantum of 2^n cells (for integer $n <$ log(HEU)) that will contain it. Larger objects are allocated the smallest multiple of HEU chunks that can contain them. When copying objects, smaller than HEU, into the new heap, the following rule is observed: "All objects in a chunk are always uniform in size." If HEU is an integral power of two, then no portion of any chunk is wasted. When allocating heap space for objects of size greater than one HEU, contiguous chunks are used.

In this refined model, chunks are categorized by the size of the objects they contain. To effectively manage this added complexity, a PE manipulates multiple $\{S, B\}$ pairs (called $\{S_1, B_1\}$, $\{S_2, B_2\}$, $\{S_4, B_4\}$, ..., and $\{S_{\text{HEU}}, B_{\text{HEU}}\}$). Initially, each PE allocates multiple chunks with S_i and B_i set to the top of each chunk.

Referring back to Figure 9, recall that shaded chunks of the heap are owned by PE(i) and non-shaded chunks are owned by other PEs. The chunks shaded as '/', in the extended model, contâin objects of some fixed size k, and are managed with a pointer pair $\{S_k, B_k\}$. Chunks shaded as '\' are either directly referenced by other pointer pairs of PE(i) (if they hold objects of size $m \neq k$), or are kept in the global pool.

Load Balancing: In the previous algorithm, it is a difficult choice to select an optimal HEU. As HEU increases, B_{global} accesses become less frequent (which is desirable, since contention is reduced); however, the average distance between S and B (in units of chunks) decreases. This means that the chance of load balancing decreases with increasing HEU.

One solution to this dilemma is to introduce an independent, constant size unit for load balancing. The load distribution unit (LDU) is this predefined constant, which is distinct from HEU[10] and enables more frequent load balancing during GC. In general, the optimized algorithm incorporates a new rule, wherein if $(B_k - S_k > LDU)$, then the region between the two pointers (i.e., the region to be scanned later) is pushed onto the global pool.

4.3.2 Evaluation

The parallel GC algorithm was evaluated for a large set of benchmark programs (from [Tick 1991] etc.) executing on a parallel KL1 emulator on a Sequent Symmetry. Statistics in the tables where measured on eight PEs with HEU=256 words and LDU=32 words, unless specified otherwise. A more detailed evaluation is given in [Imai and Tick 1991].

To evaluate load balancing during GC, we define the *workload* of a PE and the *speedup* of a system as follows:

$$\begin{aligned} \text{workload(PE)} &= \text{number of cells copied} + \\ &\quad\ \text{number of cells scanned} \\ \text{speedup} &= \frac{\sum \text{workloads}}{max(\text{workload of PEs})} \end{aligned}$$

The workload value approximates the GC time, which cannot be accurately measured because it is affected by DYNIX scheduling on Symmetry. Workload is measured in units of *cells referenced*. Speedup is calculated with the assumption that the PE with the *maximum* workload determines the *total* GC time. Note that speedup only represents how well load balancing is performed and does not take into account any extra overheads of load balancing (which are tackled separately). We also define the *ideal speedup* of a system:

$$\text{ideal speedup} = min\left(\frac{\sum \text{workloads}}{max(\text{workload for one object})}, \#\text{PEs}\right)$$

[10] We assume that HEU = kLDU, for integer $k > 0$.

	avg.	Speedup				
	WL	Size of LDU				
Benchmark	×1000	32w	64w	128w	256w	ideal
BestPath	165	7.15	7.06	6.46	6.36	8.00
Boyer	47	5.67	5.83	4.38	4.12	8.00
Cube	139	7.74	7.67	7.35	6.83	8.00
Life	101	7.10	6.86	6.31	6.29	8.00
MasterMind	4	2.50	2.48	2.58	2.48	2.87
MaxFlow	95	4.06	3.84	3.70	2.86	8.00
Pascal	5	2.67	2.91	3.45	2.77	7.25
Pentomino	3	4.34	3.34	3.67	4.21	8.00
Puzzle	17	2.63	2.84	2.58	2.61	2.92
SemiGroup	496	7.75	7.28	7.49	7.02	8.00
TP	17	2.49	2.39	2.43	2.33	2.79
Turtles	203	7.79	7.44	7.20	7.22	8.00
Waltz	32	4.38	2.92	2.31	1.64	8.00
Zebra	167	6.27	6.04	6.42	6.28	8.00

Table 4: Average Workload and Speedup (8 PEs, HEU=256 words)

Ideal speedup is meant to be an approximate measure of the fastest that n PEs can perform GC. Given a perfect load distribution where $1/n$ of the sum of the workloads is performed on each PE, the ideal speedup is n. There is an obvious case when an ideal speedup of n cannot be achieved: when a single data object is so large that its workload is greater than $1/n$ of the sum of the workloads. In this case, GC can complete only after the workload for this object has completed. These intuitions are formulated in the above definition.

Speedup: Table 4 summarizes the average workload and speedup metrics for the benchmarks. The table shows that benchmarks with larger workloads display higher speedups. This illustrates that the algorithm is quite practical. It also shows that the smaller the LDU, the higher the speedup obtained. This means there are the more chances to distribute unscanned regions, as we hypothesized.

In some benchmarks, such as MasterMind, Puzzle and TP, ideal speedup is limited (2–3). This limitation is due to an inability of PEs to cooperate in accessing a single large structure. The biggest structure in each of the benchmark programs is the *program module*. A program module is actually a first-class structure and therefore subject to garbage collection (necessary for a self-contained KL1 system which includes a debugger and incremental compiler). In practice, application programs consist of many modules, opposed to the benchmarks measured here, with only a single module per program. Thus the limitation of ideal speedup in MasterMind and Puzzle is peculiar to these toy programs.

In benchmarks such as Pascal and Waltz, the achieved speedup is significantly less than the ideal speedup. These programs create many long, flat lists. When copying such lists, S and B are incremented at the same rate. The proposed load distribution mechanism does not work well in these degenerate cases. Our method works best for deeper structures, so that B is incremented at a faster rate than S. In this case, ample work is uncovered and added to the global pool for distribution.

	LDU (words)			
Benchmark	32	64	128	256
BestPath	421.0	139.6	84.4	45.8
Boyer	208.8	131.3	24.3	12.8
Cube	609.4	241.6	96.3	55.5
Life	145.8	66.5	29.8	14.8
MasterMind	3.9	1.5	1.1	1.0
MaxFlow	211.3	75.0	37.0	10.0
Pascal	1.6	1.0	1.0	1.0
Pentomino	134.3	65.3	21.0	7.5
Puzzle	51.6	30.6	10.5	4.9
SemiGroup	1,700.7	910.8	439.3	29.6
TP	44.4	19.8	8.8	4.6
Turtles	1,427.0	640.0	314.0	136.0
Waltz	76.0	36.0	11.5	1.4
Zebra	2,127.9	920.2	467.7	222.4

Table 5: Accesses of the Global Pool (8 PEs, HEU=256 words)

Contention at the Global Heap Bottom: We analyzed the frequency with which the global heap-bottom pointer, B_{global}, is updated (for allocation of new chunks). This action is important because B_{global} is shared by all the PEs, which must lock each other out of the critical sections that manage the pointer. For instance, in Zebra (given HEU = 256 words and LDU = 32 words), B_{global} is updated 3,885 times by GCs. If B_{global} were updated whenever a single object was copied to the new heap, the value would be updated 126,761 times. Thus, the update frequency is reduced by over 32 times compared to this naive update scheme. In other benchmarks, the ratios of the other programs range from 15 to 114.

Global-Pool Access Behavior: Table 5 shows the average number of global-pool accesses made by the benchmarks, and the average number of cells referenced (in thousands) by the benchmarks per global-pool access. These statistics are shown with varying LDU sizes. The data confirms that, except for Pascal and MasterMind, the smaller the LDU, the more chances these are to distribute unscanned regions, as we hypothesized. The amount of distribution overhead is at least two orders of magnitude below the useful GC work, and in most cases, at least three orders of magnitude below.

As described above, to achieve efficient garbage collection on a shared-memory multiprocessor system, load distribution and the working set size should also be carefully considered.

4.4 Goal Scheduling in a Cluster

An efficient goal scheduling algorithm within a cluster must satisfy the following criteria:

1. no idle processing elements
2. high data locality
3. less access contention
4. no disturbance of busy processing elements

Moreover, since the KL1 language has the concept of goal priority (Section 3.1.3), goals with higher priorities within a cluster are the targets of scheduling. Notice that *Load* is the amount of work to be performed by a PE, cluster or system. Thus, load does not mean the number of goals.

No Idle Processing Elements: The aim of goal scheduling is to finish the execution of application programs earlier. Previous software simulation told us the following [Sato and Goto 1988]:

- To keep all PEs busy is the most effective way of load balancing since the goals of the KL1 language are, in general, fine-grained and have rich parallelism.
- Making the numbers of goals of each PE the same during execution does not lead to good load balancing.

Here, an idle PE means one that does not have any goals to be reduced, or one that reduces goals with lower priorities.

High Data Locality: Since a cluster is viewed as a shared-memory multiprocessor, it is important to keep the data locality high to achieve high performance. This means keeping the hit ratio of snooping caches high. In our KL1 runtime system, once argument data are allocated to a memory, the locations are not moved (only a garbage collector can move them). Hence, it is desirable that a goal that includes references to the argument data is reduced by a PE in which the cache already contains the data. Furthermore, in terms of KL1 goal reduction, suspension and resumption during unification give rise to expensive context switching. If context switching occurs frequently, the hit ratio of snooping caches decreases and, consequently, the total performance is seriously degraded.

Less Access Contention: To schedule goals properly, each PE has to access shared resources in parallel. For instance, there is a goal pool that stores goals to be reduced and priority information that must be exchanged among PEs. Since expensive mutual exclusion is required when PEs within a cluster access these shared resources, access conflicts should be decreased as much as possible.

No Disturbance of Busy Processing Elements: From the load balancing point of view, it is better to have as many idle PEs as possible involved in work associated with goal scheduling. Moreover, when an idle PE tries to find a new goal, it is desirable that the idle PE should neither interrupt nor disturb the execution of busy PEs.

Consequently, well-distributed data structures and algorithms should be designed so that these criteria are satisfied as much as possible.

4.4.1 Goal Pool

Let us consider two ways of implementing a goal pool: centralized implementation and distributed implementation. That is, one queue in a cluster or one queue for every PE. If centralized implementation is used, priority is strictly managed. However, every time a goal is picked up and new goals are stored, the access contention may occur. Thus, our KL1 implementation adopts the distributed implementation method. It turns out that transmission of goals between PEs for load balancing is required and priority is loosely managed. On the contrary, however, distributed queue management is necessarily loose for priority.

The distributed goal queues are managed using a depth-first rule to keep the data locality high. Under depth-first (LIFO) management, it is presumed that the same PE will often write and read the same data and that the number of suspensions and resumptions invoked will be less. Therefore, the cache hit ratio increases.

Further, when a PE resumes goal unification, the PE sends the goal to the queue of the PE which suspended the goal previously. This also contributes to keeping the data locality high.

As described above, since goals are accompanied with priorities, in our KL1 implementation, a PE has its own goal queues for each priority. Figure 10 shows the goal queues with priorities.

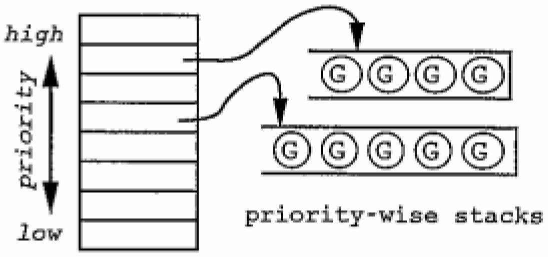


Figure 10: Goal Queue with Priorities

4.4.2 Transmission of Goals

As soon as a PE becomes or may become idle, it must take a new goal with higher priority from the queue of a PE with a small overhead to avoid going into an idle state. An idle PE triggers the transmission of a new goal.

Here, two design decisions are needed. One decision is deciding whether the PE that transmits a new goal with high priority is a request sender (idle PE) or a request receiver (busy PE). Another decision is deciding whether a new goal is to be picked from the top of a queue or the end. If an idle PE has the initiative, access contention may occur in the queue of a busy PE. If a busy PE has the initiative, the CPU time of the busy PE must be consumed. If a new goal is picked from the top of a queue, it may destroy the data locality of the busy PE's cache. If a new goal is at the end, it will often happen that the goal reduction of an idle PE is immediately suspended; the potential load of the goal may be small under LIFO management. Thus, this method may frequently trigger transmission.

The current implementation uses dedicated PIM hardware which broadcasts requests to all PEs within a cluster, in order to issue a request for a new goal to the other PEs. Each busy PE executes an event handler once a reduction and the event handler may catch the request. Then, the busy PE which catches the request first picks up the goal with the highest priority from the top of its goal queue. Our implementation should be evaluated for comparison.

4.4.3 Priority Balancing

A PE always reduces goals which belong to its local queue and have the highest priority. There are two problems; one is how to detect the priority imbalance, and the other is how to correct the imbalance by cooperating with the other PEs. Our priority balancing scheme was designed so that fewer shared resources are required and busy PEs do less work concerned with priority balancing (Figure 11). Our scheme requires only one shared variable P_a to record an average priority, and the same number of variables $I_1 \sim I_n$ as the number of PEs to record a current integral value for each PE. A current priority of each PE is represented by P_i. There are two constants, max (> 0) and min (< 0). Every PE will always calculate the integral I_i of $P_i - P_a$ along time. When $I_i > max$, the PE(i) adjusts P_a to the current P_i and resets I_i to zero. When $I_i < min$, the PE(i) issues a goal request, adjusts P_a to the priority of a transmitted goal, and resets I_i to zero. The mechanism of the goal transmission described above is used as well, since the goal with the highest PE priority is picked up. More details on this algorithm are described in [Nakagawa *et al.* 1989].

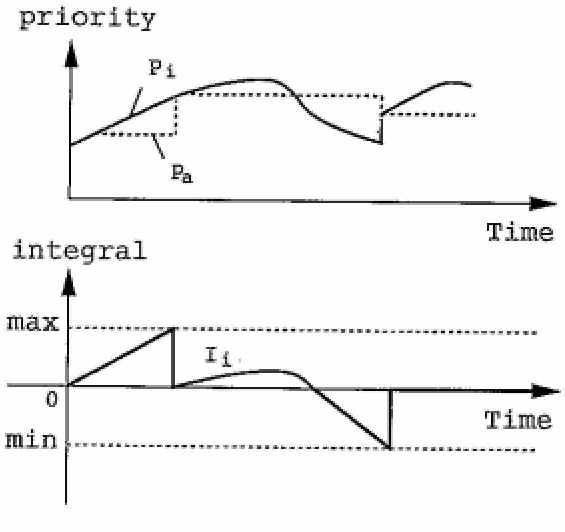


Figure 11: Priority Balancing Scheme

The features of this scheme are as follows. The calculation of the integral reduces the frequency of shared resource P_a updating and busy PEs do some work only when $I > max$.

The disadvantages are as follows. It may happen that the priority of a transmitted goal is even lower, that P_a decreases unreasonably, and that the frequency of the high-priority goal transmission decreases. Our priority balancing scheme utilizes the goal transmission mechanism (Section 4.4.2), which does not always transfer the goal with the most appropriate priority. Accordingly, a load imbalance may be sustained for a while. How well this method works depends on the priority of the goals transmitted upon requests. In other words, there is a tradeoff between loose priority management and the frequency of high-priority goal transmission. Further, in this scheme, a busy PE (a PE satisfying $I_i > max$) has to write its current priority P_i to the shared variable P_a. This may cause access conflict and disturb the busy PE.

A new scheme which we will design should overcome these problems. However, we think that calculation of the integral along time is essential even in new schemes.

4.5 Meta Control Facilities

When designing the implementation for a shoen, we assume that the following dynamic behavior applies in the KL1 system:

- Shoen statuses change infrequently.
- Shoen operations are not executed immediately but within a finite time.
- Messages transferred are possibly overtaken in the inter-cluster network.

Under these assumptions, our implementation must satisfy the following requirements:

- The less inter-cluster messages the better.
- No bottleneck appears; algorithms and protocols that do not frequently access shoen records and foster-parent records are desirable.
- The processing associated with meta control should not degrade the performance of reduction.

Many techniques realizing a shoen have been developed to achieve high efficiency. This section concentrates on execution control and resource management.

From now on, stream messages on the control and report streams for communication to the outside are represented in a typewriter typeface, such as `start`, `add_resource`, and `ask_statistics`.

4.5.1 Execution Control

This section describes schemes for implementing the functions for execution control. Schemes (1) $\sim$ (2) are effective in a shared-memory environment (intra-cluster). Schemes (3) $\sim$ (5) are effective in a distributed-memory environment (inter-cluster).

(1) Change of Foster-parent Status: Since goal reduction cannot be started when the status of foster-parent which the goal belongs to is not *started*, imprudent implementation needs to check the status of a foster-parent before every goal reduction. To avoid such frequent checking, a status change of the foster-parent is notified by the interruption mechanism. When a cluster receives a message that changes a foster-parent's status to non-executable, an interruption is issued to every PE in the cluster. When a PE catches the interruption, the PE checks to see if the current goal belongs to the target foster-parent. If so, then the foster-parent is to be stopped and the PE suspends execution of the current goal and starts to reduce the goal of the other active foster-parent. Otherwise, the PE continues the reduction. Since the newly scheduled goal is supposed to belong to the other foster-parent, the context of the goal reduction [11] must be switched, too.

The assumption that the status of a foster-parent is switched infrequently implies that interruptions happen rarely. Thus, an advantage of the scheme is that the ordinary reduction process rarely suffers from foster-parent checking.

(2) Foster-parent Termination Detection: To detect the termination of a foster-parent efficiently, a counter called *childcount* is introduced. The childcount represents the sum of both the number of goals and the number of shoens which belong to the foster-parent. When the childcount of a foster-parent reaches zero, all goals of the foster-parent are finished.

The childcount area is allocated in a foster-parent record, and all PEs in a cluster must access the area. Since this counter must be updated whenever a goal is created or terminated, frequent exclusive updating of this counter might become a bottleneck. To reduce such an access contention, the cache area of the childcount is allocated on each PE. The operations go as follows. At first, a counter is allocated on the childcount cache of each PE, initialized with a value zero. Every time a new goal is spawn, the counter is incremented, and the counter is decremented upon the end of goal reduction. When the reduction of a new goal whose foster-parent differs from the previous one begins, the current foster-parent should be switched. That is, the value of the counter on the childcount cache is brought back to the previous foster-parent record, and the counter is reinitialized. The foster-parent terminates when it detects that the counter on the foster-parent record is zero.

This scheme is expected to work efficiently if foster-parents are not changed often.

(3) Point-to-point Message Protocol: Basically, message protocols based on point-to-point communication between a shoen and a foster-parent are not designed on the basis of broadcasting [Rokusawa *et al.* 1988]. If almost all clusters always contain foster-parents of a shoen, protocols based on broadcast are taken into account. However, the current implemetation does not assume this, although it depends on applications. Therefore, it is inefficient to broadcast messages to all clusters in the system every time. Then, a shoen provides a table that indicates whether or not its foster-parent exists in a cluster corresponding to the table position. The table is maintained by receiving foster-parent creation and termination messages from the other clusters. Accordingly, a shoen can send messages only to the clusters where its foster-parents reside.

(4) Lazy Management of Foster-parent: A shoen controls its foster-parents by exchanging messages, such as start/stop messages. However, these messages may overtake, and, thus, a foster-parent may go into the incorrect states. For the stats to be correct and to minimize the maintenance cost, received start/stop messages are managed by a counter. If a start message arrives, the foster-parent increments the counter. If a stop message arrives, the foster-parent decrements the counter. Then, when the counter value crosses zero, the foster-parent changes the execution status properly.

(5) Shoen Termination Detection: To detect the termination of a shoen efficiently, a Weighted Throw Count (WTC) scheme was introduced [Rokusawa *et al.* 1988] [Rokusawa and Ichiyoshi 1992]. This scheme is also an application of the weighted reference count scheme [Watson and Watson 1987][Bevan 1989]. Logically, a shoen is terminated when there are no foster-parents. However, this is not correct enough to maintain the number of foster-parents, since goals thrown by a foster-parent may be transferred in the network. Thus, a foster-parent lets both all goals to be thrown and all messages between a shoen and foster-parents to have a portion of the foster-parent's weight. On terminating a foster-parent, all foster-parent weights are returned to

[11] A childcount cache and a resource cache.

the shoen. If the foster-parent terminated at message arrival, the messages from the shoen are also sent back to the shoen to keep its weight. Then, when all weights are returned to the shoen, the shoen terminates itself. An advantage of this scheme is that it is free from sending acknowledgement messages.

Thus, since a shoen must not continue to lock shared resources in this scheme until an acknowledgement returns, the scheme can reduce not only the network traffic but can also alleviate mutual exclusion.

4.5.2 Resource Management

As described above, a shoen is also used as a unit for resource management. In the KL1 language, the reduction time is regarded as the computation resource. The shoen consumes the supplied resources while shifting the resources. Moreover, since a shoen works in parallel, lazy resource management is inevitable, like in the shoen execution control (Section 4.5.1).

A shoen has a limited amount of resources which it can consume. Upon exceeding the limit, goals in the shoen cannot be reduced. When a runtime system detects that the total amount of consumed resources so far is approaching the limit, a `resource_low` message is automatically issued on the shoen's report stream. The shoen stops its execution with its resources exhausted. On the other hand, the `add_resource` message on the control stream raises the limit and the shoen can utilize the resource up to the new limit. Furthermore, a shoen which accepts the `ask_statistics` message reports the current resources consumed so far.

This section describes our resource management implementation schemes.

(1) Distributed Management: The scheme is briefly described below. Figure 12 shows the resource flow between a shoen and its foster-parents.

A shoen has a limit value, which indicates that the shoen can consume resources up to the limit. Initially, the resource limit is zero. Only the `add_resource` message can raise the limit. When a shoen receives the `add_resource` message, the shoen requests new resources to the above foster-parent by a value within the limit value designated by the `add_resource` message. Here, we also call this foster-parent the *parent* foster-parent. Notice that a shoen and its parent foster-parent reside in the same cluster, and, thus, the operation for the resource request is implemented by read and write operations on a shared memory.

After a shoen has got new resources from its parent foster-parent, the shoen further supplies resources to its foster-parents which requested resources by the supply_resource message across clusters. Moreover the supplied resources may be supplied to the descendant shoens and foster-parents. Then, those foster-parents consume the supplied resources. The shoen has a buffer for the resources; the excessive resources returned from terminated foster-parents are stored in the shoen buffer. When the remaining resources of a foster-parent are going to run out, a resource request message is sent to the above shoen. If the shoen cannot afford to supply the requested resources, the shoen issues the `resource_low` message on its report stream. Otherwise, if the shoen can afford and has sufficient resources in the buffer, the resources are supplied to the foster-parent immediately. If there are insufficient resources, the shoen requests new resources within the current limit value from its parent foster-parent. As described here, the resource buffer of a shoen can prevent the message from being issued more frequently than necessary.

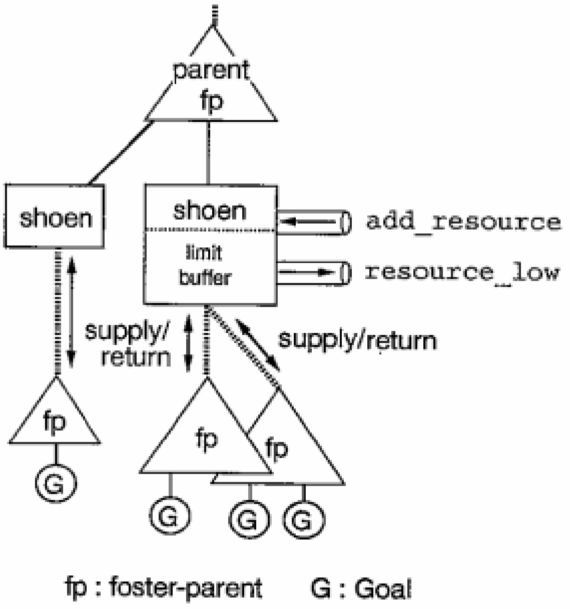


Figure 12: Resource Flow Between a Shoen and its Foster-parents

If the resources of the foster-parent are exhausted, goal reduction stops. Then, the scheduled goals are hooked on to the foster-parent record, in preparation for rescheduling when new resources are supplied from the shoen.

Furthermore, each PE has a resource cache area for the foster-parent, and, hence, a counter is actually decremented every time a goal is reduced. This mechanism is similar to the childcount mechanism (Section 4.5.1). However, when the foster-parent of a goal to be reduced alters, the caches on PEs must be brought back to the foster-parent record.

(2) Resource Statistics: While the system enjoys lazy resource management, it gets harder to collect resource information over the entire system. A shoen receives the `ask_statistics` message, which reports the current total consumed resources.

The scheme used to collect the information is described. A shoen issues inquiry messages to each foster-parent. When an inquiry message arrives at a foster-parent, the foster-parent informs each PE of this using the interruption mechanism. This portion is similar to the mechanism of Section 4.5.1 (1). The PEs which catch the interruption check if the current goals belong to the target foster-parent. If so, the PE puts the resource on the cache back to the foster-parent record. When all corresponding PEs have been put back, the subtotal resource on the foster-parent appears. If not, the PEs do nothing and reduction continues. Then, the foster-parent reports the subtotal to the shoen and re-distributes some resources back to the PEs. As a result, the PEs resume goal reduction.

We assume that the ask_statistics message is issued infrequently. This scheme works well.

(3) Point-to-point Resource Delivery: The destination of new resources when a shoen receives resource request messages from its foster-parents is a design decision. It must be decided whether the shoen delivers the new resources only to the foster-parents which have requested them, or delivers them to all foster-parents. A protocol based on broadcast may be preferable when the foster-parents in nearly all clusters always possess the same amount of resources and consume them at the same speed. The current method is similar to one in Section 4.5.1 (3).

Our assumptions we based on an experience of the Muti-PSI system. Goal scheduling within a cluster, however, differs and there is no guarantee that every cluster has the foster-parent of the shoen. Therefore, in the current implementation method the shoen sends the resource supply message just to the clusters which have sent resource request messages.

4.6 Intermediate Instruction Set

The KL1 compiler for PIM has two phases. The first phase compiles a KL1 program into an intermediate instruction code; the instruction set is called KL1-B. The second phase translates the intermediate code into a native code. KL1-B is designed for an *abstract KL1 machine* [Kimura and Chikayama 1987], interfacing between the KL1 language and PIM hardware, just as in Warren Abstract Machine [Warren 1983] of Prolog.

KL1-B for PIM is extended from KL1-B for Multi-PSI to efficiently exploit the PIM hardware.

4.6.1 Abstract KL1 machine

The *abstract KL1 machine* is simple virtual hardware to describe a KL1 execution mechanism. It has a single PE with a heap memory and basically expresses the inside execution of a PE. However, every KL1-B instruction implicitly supports multi-PE processing. Further, some KL1-B instructions are added for inter-*cluster* processing.

A goal is represented by a *goal record* on a heap. The *goal record* consists of arguments and an execution environment which includes the number of arguments and the address of the predicate code. A ready goal is managed in the *ready goal pool* which has entries for each priority. Each entry indicates a linked stack of *goal records*. Suspended goals are *hooked* on the responsible variable.

Each data word consists of a value part, a type part and a MRB part [Chikayama and Kimura 1987]. An MRB part is valid, if the value part is a pointer, and indicates whether its object is single-referenced or multiple-referenced. It is used for incremental garbage collection and destructive structure updating.

4.6.2 Overview of KL1-B

The intermediate instruction set KL1-B was designed according to the following principles:

- Memory based scheme — goal arguments are basically kept on a goal record at the beginning of a reduction, and each of them is read onto a register explicitly just before it is demanded. Thus, almost all registers are used temporarily (Section 4.6.3).
- Optimization using the MRB scheme — some instructions to reuse structures are supported to alleviate execution cost (Section 4.6.4).
- Clause indexing — the compiler collects the clauses which test the same variables, and compiles them into an instruction module. Then, all guard parts of a predicate are compiled as one into the code with branch instructions forming a tree structure (Section 4.6.5).
- Each body is compiled into a sequence of instructions which run straight ahead without branching.

The basic KL1-B instruction set is shown in Table 6.

4.6.3 Memory Based Scheme

The Multi-PSI system executes a KL1 program using the *register based scheme* — all arguments of the current goal are loaded onto *argument registers* before reduction begins, just as WAM does for Prolog.

Here, let us compare the following two methods in terms of the argument manipulation cost:

- In the *memory based scheme*, the arguments referred to in the reduction are loaded and the modified arguments are stored at every reduction. There is no cost for goal switching.
- In the *register based scheme*, all arguments of the swapped out goal are stored and all arguments of the swapped in goal are loaded at every goal switching.

Table 6: Basic KL1-B Instruction Set

KL1-B Instruction		Specification
For passive unification:		
load_wait	*Rgp,Pos,Rx,Lsus*	Read a goal argument onto *Rx* and check binding.
read_wait	*Rsp,Pos,Rx,Lsus*	Read a structure element onto *Rx* and check binding.
is_atom/integer/list/...	*Rx,Lfail*	Test data type of *Rx*.
test_atom/integer	*Rx,Const,Lfail*	Test data value of *Rx*.
equal	*Rx,Ry,Lsus,Lfail*	General unification.
suspend	*Lpred,Arity*	Suspend the current goal
For argument/element preparation:		
load	*Pgp,Pos,Rx*	Read a goal argument onto *Rx*.
read	*Rsp,Pos,Rx*	Read a structure element onto *Rx*.
put_atom/integer	*Const,Rx*	Put the atomic constant onto *Rx*.
alloc_variable	*Rx*	Allocate a new variable and put the pointer onto *Rx*.
alloc_list/vector	(*Arity,*)*Rx*	Allocate a new list/vector structure and put the pointer onto *Rx*.
write	*Rx,Rsp,Pos*	Write *Rx* onto a structure element.
For incremental garbage collection:		
mark	*Rx*	Mark MRB of *Rx*.
collect_value	*Rx*	Collect the structure recursively unless its MRB is marked.
collect_list/vector	(*Arity,*)*Rx*	Collect the list structure unless its MRB is marked.
reuse_list/vector	(*Arity,*)*Rx*	*collect_list/vector* + *alloc_list/vector.*
For active unification:		
unify_atom/integer	*Const,Rx*	Unify *Rx* with the atomic constant.
unify_bound_value	*Rsp,Rx*	Unify *Rx* with the newly allocated structure.
unify	*Rx, Ry*	General unification.
For goal manipulation and event handling:		
collect_goal	*Arity,Rgp*	Reclaim the *goal record.*
alloc_goal	*Arity,Rgp*	Allocate a new *goal record.*
store	*Rx,Rgp,Pos*	Write *Rx* onto a goal argument.
get_code	*CodeSpec,Rcode*	Get the code address of the predicate onto *Rcode*.
push_goal	*Rgp,Rcode,Arity*	Push the goal to the current priority entry of *ready goal pool.*
push_goal_with_priority	*Rgp,Rcode,Rprio,Arity*	Push the goal to the specified priority entry of *ready goal pool.*
throw_goal	*Rgp,Rcode,Rcls,Arity*	Throw the goal to the specified *cluster.*
execute	*Rcode,Arity*	Handle the event if it occurrs and execute the goal repeatedly.
proceed		Handle the event if it occurrs and take a new goal from *ready goal pool* to start the new reduction.

Some arguments may be moved between registers at every reduction.

Therefore, the *memory based scheme* is better than the *register based scheme* when

- Goal switching occurs frequently.
- A goal has many arguments.
- A goal does not refer to many arguments in a reduction.

Actually, these cases are expected to be seen often in large KL1 programs. Thus, we have to verify the *memory based scheme* with many practical KL1 applications.

Additionally, the number of goal arguments is limited to the number of *argument registers* — 32 in the case of Multi-PSI. This limitation is too tight and is not favorable to KL1 programmers. The *memory based scheme* can alleviate this limitation to some extent. On the other hand, the naive *memory based scheme* necessarily writes back all arguments to the *goal record*, even if tail recursion is employed. Since this is very wasteful, an optimization to keep frequently referenced arguments on registers is mandatory during tail recursion.

4.6.4 Optimization

Two optimization techniques are introduced: tail recursive optimization and the reuse of data structures. We can describe these using the following sample codes.

- source code:

```
app([H|L],T,X) :- true | X=[H|Y], app(L,T,Y).
app([ ]    ,T,X) :- true | X=T.
```

- intermediate code:

```
app_entry:
  load               CGP, 0, R1        % Load up
```

```
    load               CGP, 2, R2      %  arguments
  app_loop:
    wait               R1, sus_or_fail
    is_list            R1, next
    commit
  ⋆ read               R1, car, R3     % H
    read               R1, cdr, R4     % L
    reuse_list         R1
  ⋆ write              R3, R1, car     % H
    alloc_variable     R5              % Y
    write              R5, R1, cdr
    unify_bound_value  R1, R2
    move               R4, R1
    move               R5, R2
    execute_tro        app_loop
          ⋮
  next:
    is_atom            R1, sus_or_fail
    test_atom          [], R1
    commit
    load               CGP, 1, R3      % T
    unify              R3, R2
    collect_goal       3, CGP
    proceed
  sus_or_fail:
    store              R1, CGP, 0      % Write back
    store              R2, CGP, 2      %  arguments
    suspend            app_entry, 3
```

Tail Recursive Optimization: Some instructions are added for this optimization. *Wait* tests if an argument on a register is instantiated. *Move* prepares arguments for the next reduction. *Execute_tro* executes a goal while some arguments are kept on registers.

In the above source code, the first and third arguments of the first clause are used in tail recursion. These arguments are loaded at the beginning of the reduction by the *load* instructions which are placed before the tail recursive loop. There is no need to write them into the *goal record* during tail recursion. However, they must be written back to the *goal record* explicitly before, say, switching the goal caused by the *suspend* instruction. Since the second argument is not used in tail recursion, it is kept on the *goal record* until it is referred to in the second clause.

In this example, two *write* instructions and two *read* instructions are replaced with two *move* instructions. Thus, by assuming a cache hit ratio of 100 %, this optimization can save two steps on each recursion loop.

Reuse of Data Structures: KL1-B for PIM supports the reuse of data structures. The *reuse_list* and *reuse_vector* instructions realize this. These instructions reuse an area in a heap on which the structure unified in a guard part was allocated, but, only if the MRB of the reference to the area is not marked. However, the area for the element data of the reused structure is not reused.

In KL1 applications, it often happens that the areas of reclaimed structures can be reused for successive allocation. This is frequent in programs for list processing and programs written in message driven programming. In the sample codes in Section 4.6.3, element H of the passive-unified list [H|L] is used as element H of the new list [H|Y], and is read and written by the instructions marked with stars ("⋆"). However, if the MRB of the *passive-unified* list is not marked, element H can actually be used in the new list as is, and, therefore, *read* and *write* instructions can be eliminated.

Therefore, the following new optimized instructions are introduced:

```
reuse_list_with_elements      Reg, [F_car|F_cdr]
reuse_vector_with_elements    Arity,Reg,{F_0,F_1,...,F_n}
```

These instructions do nothing when the MRB of the structure pointer on *Reg* is not marked. If marked, they allocate a new structure, copy specified elements on the structure referenced by *Reg* to the new structure, and put the pointer to the new structure onto *Reg*. Thus, reuse of data structures reduces the number of memory operations and, accordingly, keeps the size of the working set small.

Sample code is shown as follows:

- optimized intermediate code:

```
       ⋮
  app_loop:
    wait                      R1, sus_or_fail
    is_list                   R1, next
    commit
    read                      R1, cdr, R4       % L
    reuse_list_with_elements  R1, [1|0]
    alloc_variable            R5                % Y
    write                     R5, R1, cdr
    unify_bound_value         R1, R2
    move                      R4, R1
    move                      R5, R2
    execute_tro               app_loop
       ⋮
```

In this code, *reuse_list* and instructions marked with stars ("⋆") are replaced with the *reuse_list_with_elements* instruction. The second argument [*1*|*0*] specifies that the head element has to be copied if the MRB of the list pointer on *R1* is marked. If the MRB is not marked, it does nothing and is equal to *nop*. Therefore, only the following *write R5,R1,cdr* instruction can allocate the list structure [H|Y]; the instruction works like the *rplacd* function in LISP. Consequently, in this example, reuse optimization can save one *read* and one *write* instructions and is worth approximately two machine steps.

4.6.5 Clause Indexing

The KL1 language neither defines the testing order for the clause selection nor has the backtracking mechanism. Thus, to attain quick suspension detection and quick clause selection, the compiler can arrange the testing order of KL1 clauses; this is called clause indexing. At first, the compiler collects the clauses which test the same variable, and compiles the clauses into shared instructions. Most of these work as test-and-branch instructions with branch labels occurring in the instruction codes. All guard parts of a predicate are, then, compiled into a tree structure of instructions.

Our KL1 programming experiences up to now have told as that a clause is infrequently selected according to the type of argument but is often selected according to the value. Further, even if multi-way switching of KL1-B instructions on data types is introduced, these KL1-B instructions are eventually implemented by a combination of native binary branch instructions, in general. Consequently, we decided that KL1-B does not provide a multi-way switching instruction on data types, but just binary-branch KL1-B instructions on a data type. Additionally, KL1-B provides a multi-way jump instruction on the value of an instantiated variable.

Two instructions are added for multi-way jump on a value:

switch_atom *Reg*, $[\{X_1,L_1\},\{X_2,L_2\}, \ldots ,\{X_n,L_n\}]$
switch_integer *Reg*, $[\{X_1,L_1\},\{X_2,L_2\}, \ldots ,\{X_n,L_n\}]$

Switch_atom is used for multi-way switching on an atom value, and *switch_integer* is used for multi-way switching on an integer value. They test the value on the register *Reg*, and if it is equal to the value X_i, a branch to the instruction specified by the label L_i occurs. Since the internal algorithm implementing these switching instructions is not defined in KL1-B, the translator to a native code may choose the most suitable method for switching.

The current KL1-B instruction set was designed under several assumptions in terms of KL1 programs. Thus, we have to investigate how correct our assumptions are and how effective our KL1-B instruction set is.

5 Conclusion

This paper discussed design and implementation issues of the KL1 language processor. PIM architecture differs from Multi-PSI architecture because of its loosely-coupled network with messages possibly overtaken, and because of its cluster structure (i.e. its shared-memory multiprocessor portion). These differences greatly influence the KL1 language processor and are essential to parallel and distributed implementation of the KL1 language. Several of the implementation issues focused on in this paper are more or less associated with these features. Our implementation is a solution to this situation. ICOT has been working on these implementation issues intensively for the past four years, since 1988.

In this paper, we began by making several assumptions and, then, tailored our implementation to them. The assumptions came from our experiences based on the Multi-PSI system. Thus, we have to evaluate our implementation, accumulate experiences on our system, and verify the appropriateness of the assumptions. Hence, we will be able to reflect our results in the KL1 language processor of the next generation. In this development cycle, the systematic design concept is effective, and the concept yields the high modularity of a language processor. It turns out to be easy to improve and highly testable.

Our KL1 language processor is presented on the PIM systems (*PIM/p*, *PIM/c*, *PIM/i*, *PIM/k*), which are being demonstrated at FGCS'92.

Acknowledgment

We would like to thank all ICOT researchers and company researchers who have been involved in the implementation of the KL1 language so far, especially, Dr. Atsuhiro Goto, Mr. Takayuki Nakagawa, and Mr. Masatoshi Sato. We also wish to thank the R&D members of Fujitsu Social Science Laboratory. Through their valuable contributions, we have achieved a practical KL1 language processor. Thanks also to Dr. Evan Tick of University of Oregon, for his great efforts in evaluating the parallel garbage collector with us. We would also like to thank Dr. Kazuhiro Fuchi, Director of ICOT Research Center, and Dr. Shunichi Uchida, Manager of Research Department ICOT, for giving us the opportunity to develop the KL1 language processor.

References

[Baker 1978] H. G. Baker. List Processing in Real Time on a Serial Computer. *Communications of the ACM*, 21(4), 1978, pp.280–294.

[Bevan 1989] D. I. Bevan. Distributed Garbage Collection Using Reference Counting. *Parallel Computing*, 9(2), 1989, pp.179–192.

[Chikayama *et al.* 1988] T. Chikayama, H. Sato and T. Miyazaki. Overview of the Parallel Inference Machine Operating System PIMOS. In *Proc. of the Int. Conf. on Fifth Generation Computer Systems*, ICOT, Tokyo, 1988, pp. 230–251.

[Chikayama and Kimura 1987] T. Chikayama and Y. Kimura. Multiple Reference Management in Flat GHC. In *Proc. of the Fourth Int. Conf. on Logic Programming*, 1987, pp.276–293.

[Crammond 1988] J. A. Crammond. A Garbage Collection Algorithm for Shared Memory Parallel Processors. *Int. Journal of Parallel Programming*, 17(6), 1988, pp.497–522.

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

[Halstead 1985] R. H. Halstead Jr. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4), 1985, pp.501–538.

[Ichiyoshi *et al.* 1988] N. Ichiyoshi, K. Rokusawa, K. Nakajima and Y. Inamura. A New External Reference Management and Distributed Unification for KL1. *New Generation Computing*, Ohmsha Ltd. 1990, pp.159–177.

[ICOT 1st Res. Lab. 1991] ICOT 1st Research Laboratory. Tutorial on VPIM Implementation. *ICOT Technical Memorandum*, TM-1044, 1991 (*In Japanese*).

[Imai *et al.* 1991] A. Imai, K. Hirata and K. Taki. PIM Architecture and Implementations. In *Proc. of Fourth Franco Japanese Symposium*, ICOT, Rennes, France, 1991.

[Imai and Tick 1991] A. Imai and E. Tick. Evaluation of Parallel Copying Garbage Collection on a Shared-Memory Multiprocessor. *ICOT Technical Report*, TR-650, 1991. (To appear in IEEE Transactions on Parallel and Distributed Systems)

[Inamura *et al.* 1988] Y. Inamura, N. Ichiyoshi, K. Rokusawa and K. Nakajima. Optimization Techniques Using the MRB and Their Evaluation on the Multi-PSI/V2. In *Proc. of the North American Conf. on Logic Programming*, 1989, pp. 907–921 (also *ICOT Technical Report*, TR-466, 1989).

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

[Nakagawa *et al.* 1989] T. Nakagawa, A. Goto and T. Chikayama. Slit-Check Feature to Speed Up Interprocessor Software Interruption Handling. In *IPSJ SIG Reports*, 89-ARC-77-3, 1989 (*In Japanese*).

[Nakajima *et al.* 1989] K. Nakajima, Y. Inamura, N. Ichiyoshi, K. Rokusawa and T. Chikayama. Distributed Implementation of KL1 on the Multi-PSI/V2. In *Proc. of the Sixth Int. Conf. on Logic Programming*, 1989, pages 436–451.

[Nishida *et al.* 1990] K. Nishida, Y. Kimura, A. Matsumoto and A. Goto. Evaluation of MRB Garbage Collection on Parallel Logic Programming Architectures. In *Proc. of the Seventh Int. Conf. on Logic Programming*, 1990, pages 83–95.

[Rokusawa *et al.* 1988] K. Rokusawa, N. Ichiyoshi, T. Chikayama and H. Nakashima. An Efficient Termination Detection and Abortion Algorithm for Distributed Processing Systems. In *Proc. of the 1988 Int. Conf. on Parallel Processing*, Vol. 1 Architecture, 1988,pp.18–22.

[Rokusawa and Ichiyoshi 1992] K. Rokusawa and N. Ichiyoshi. A Scheme for State Change in a Distributed Environment Using Weighted Throw Counting. In *Proc. of Sixth Int. Parallel Processing Symposium*, IEEE, 1992.

[Sato and Goto 1988] M. Sato and A. Goto. Evaluation of the KL1 Parallel System on a Shared Memory Multiprocessor. In *Proc. of IFIP Working Conf. on Parallel Processing*, 1988, pp. 305–318.

[Takagi and Nakase 1991] T. Takagi and A. Nakase, Evaluation of VPIM: A Distributed KL1 Implementation – Focusing on Inter-cluster Operations –, In *IPSJ SIG Reports*, 91-ARC-89-27, 1991 (*In Japanese*).

[Taki 1992] K. Taki. Parallel Inference Machine PIM. In *Proc. of the Int. Conf. on Fifth Generation Computer Systems*, 1992.

[Tick 1991] E. Tick. *Parallel Logic Programming*. Logic Programming, MIT Press, 1991.

[Ueda and Chikayama 1990] K. Ueda and T. Chikayama. Design of the Kernal Language for the Parallel Inference Machine. *The Computer Journal*, (33)6, 1990, pp.494–500.

[Warren 1983] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, Articial Intelligence Center, SRI, 1983.

[Watson and Watson 1987] P. Watson and I. Watson. An Efficient Garbage Collection Scheme for Parallel Computer Architectures. In *Proc. of Parallel Architectures and Languages Europe*, LNCS 259, Vol.II, 1987, pp.432–443.