TR-349

# Evaluation of the KL1 Parallel System on a Shared Memory Multiprocessor

by

M. Sato & A. Goto

March, 1988

# Evaluation of the KL1 Parallel System on a Shared Memory Multiprocessor

Masatoshi SATO* Atsuhiro GOTO

Institute for New Generation Computer Technology (ICOT) †

Abstract

This paper presents the design decisions for the parallel implementation of an AND-parallel logic programming language, KL1, on a shared memory multiprocessor. To obtain high performance in parallel systems on a shared memory multiprocessor, it is necessary to mimimize the synchronization overhead and to use the processing power fully. The KL1 parallel system introduces independent scheduling queues with a depth-first scheduling scheme and an on-demand load balancing mechanism for realizing these requirements. An evaluation and detailed analysis of the design decisions are also presented. Substantial speedup can be obtained from several benchmarks according to their potential parallelism. This paper shows that on-demand load distribution and independent scheduling queues are efficient for the implementation of KL1 on shared memory multiprocessors.

## 1 Introduction

ICOT is conducting research and development of the parallel inference machine, PIM [4]. The PIM target language is *Kernel Language 1 (KL1)*, which is based on the AND-parallel logic programming language, Guarded Horn Clauses (GHC) [20,21]. Logic programming has been studied because of its theoretical opportunities for parallelism. Many research groups have been studying parallel implementation based on AND-parallelism [18,6,3] or OR-parallelism [23,5,2,19]. The reason for choosing AND-parallel logic programming as the target language is that the PIM target language must be able to describe not only artificial intelligence (AI) application programs but also the operating systems which control parallel processes. OR-parallel logic programming languages, such as pure Prolog, cannot describe the operating systems which manage the overall processing of AI programs. AND-parallel logic programming languages, such as Concurrent Prolog (CP) [15] and GHC, can easily describe the control of concurrent processes [16].

The PIM has a hierarchical structure with a cluster concept. Each cluster consists of eight or more processing elements (PEs) which communicate through shared memory (SM) over a common bus. The clusters are connected by a switching network. Two kinds of KL1 parallel implementation models, the shared heap model [12] for intra-cluster communication and the message oriented model [7,17] for inter-cluster communication, have been studied in ICOT, because the PIM's configuration

*CSNET: msato%icot.jp@relay.cs.net, ARPA: msato%icot.uucp@eddie.mit.edu, UUCP: ihnp4!kddlab!icot!msato

†Mitakokusai Building 21F, 1-4-28 Mita, Minato-ku, Tokyo, 108, JAPAN

warrants two approaches. This paper focuses on the shared heap model. Both implementation models will be integrated in the PIM global architecture.

The most important issues in the implementation of the KL1 parallel system on the shared memory are to ensure the following:

- Efficient exclusive data access
- Good load balancing to use processing power fully
- Good scheduling to minimize synchronization overhead

To enable efficient exclusive data access, we reported several important design issues for the KL1 parallel system in [12] by using a software simulator on a sequential machine. However, this software simulator was not enough to study the other issues. To study load balancing and scheduling, it is necessary to support accurate timing on a genuine multiprocessor. Therefore, we implemented the KL1 parallel system on a genuine multiprocessor, Balance 21000 [14], in the C language with some extended functions for parallel execution.

Section 2 outlines the abstract execution of KL1 with major data structures and gives important design decisions to extend the parallel system and some issues in its implementation. Section 3 discusses the characteristics of benchmark programs which are used in the following evaluations. Section 4 gives some detailed evaluations by the KL1 parallel system which is implemented on Balance 21000.

## 2 Implementation of KL1 Parallel System

### 2.1 Brief Introduction to KL1

KL1 is a parallel logic programming language based on GHC [20]. A KL1 program is a finite set of guarded Horn clauses of the following form:

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

where $H$, $G_i$, and $B_i$ are called the *clause head, guard goals,* and *body goals.* The operator, |, is called a commitment operator. The part of a clause preceding | is called the *passive-part* (or *guard*), and that following it is called the *active-part* (or *body*). A guarded clause with no head is a goal clause, as in Prolog.

Execution of a KL1 program proceeds by reducing a given goal clause to the empty clause, so it is natural to regard the processing mechanism of KL1 as *reduction.* Figure 1 shows the abstract execution features of KL1 goal reduction.

#### (1) Data and control structures and execution control

Parallel KL1 goals are represented by *goal-records* and their environments. *Goal-records* include atomic goal arguments or pointers to their environments consisting of logical variable cells or structures in the shared (heap) area. The reducible *goal-records* are stored in a *ready-queue.* Clauses in KL1 programs are compiled to KL1-B [8] just as Prolog is compiled to the WAM [22]. The PE dequeues a *goal-record* from a *ready-queue*, then performs goal reduction by executing KL1-B instructions, accessing the goal environment. Some goals are waiting for the instantiated values of variable cells to synchronize with other parallel goals. Such *goal-records* are *bind-hooked* with variable cells by *suspension-records.* The

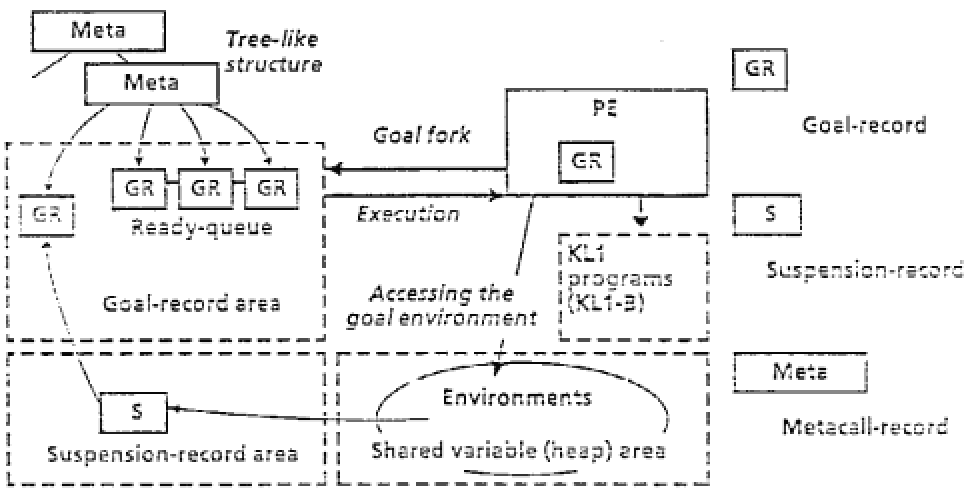

Figure 1: Abstract Features of KL1 Goal Reduction

*metacall-records* form a tree-like structure, whose leaves are the *goal-records*, to manage their logical results (success or failure).

### (2) Execution of the passive-part

In the *execution of the passive-part*, each candidate clause is tested sequentially by head unification and guard evaluation to choose one clause whose body goals will be executed. When the PE finishes all guard tests without choosing a clause, the PE checks whether this goal has failed or been suspended. This check is done by the existence of stacked variables, i.e., in the unification of the *passive-part*, uninstantiated caller variables are stacked. If the stack is empty, the PE knows that this goal reduction has failed. If it is not empty, this goal must wait until one of the variables in the stack is instantiated. Therefore, the *suspend operation* provides a link between the variables and the *goal-record* in order to activate the suspended goal immediately after one of the variables is instantiated.

### (3) Execution of the active-part

In the *execution of the active-part*, there are two kinds of operation, *body unification* and *body goal fork*.

After one clause is chosen by the execution of the *passive-part*, the PE executes the *body unifications* in the selected clause body. When the PE instantiates the variable with the *suspension-record* in this *body unification*, the PE finds the suspended goal and enqueues it to the *ready queue*.

If the selected clause body includes any user goals, the PE creates the new *goal-records* and enqueues them into the *ready queue* as new reducible goals and updates the *children counter* of the parent *metacall record*. At this point, these new *goal records* are linked to the *metacall record* where the reduced goal was linked to form a goal tree. This operation is called *body goal fork*.

## 2.2 Design Decisions for the KL1 Parallel System

### 2.2.1 General Considerations

This section focuses on parallel goal reductions by multiprocessors with shared memory, where each PE performs goal reduction in parallel, communicating through shared variables in shared memory.

The major advantage of using shared memory is the reduction in communication overhead among PEs compared with a message oriented model [7,3]. Using exclusive data access by lock and unlock, goal reduction (see Section 2.1) can be extended to a parallel mechanism. However, such a simple extension is not enough to obtain high performance [12,9], because the following problems in parallel execution on a shared memory machine also need to be solved.

- Data management to reduce the number of lock manipulations
- Distribution methods to realize a good load balance

#### 2.2.2 Separate of Data Management and Data Sharing

The parallel execution mechanism should use local control structures as much as possible to reduce common bus traffic [9], even though PEs can access all of the shared memory. In other words, KL1 goals and control structures should be *logically* treated as local data structures. Therefore, we decided to use a local ready queue on each PE, so that each PE can schedule a reduced goal independently. In addition, each PE manages its own free memory which is allocated for data structure, such as goal environments and suspension-records [12]. From this management, exclusive memory access for goal environments is restricted to when a PE instantiates an uninstantiated variable. This is because it is not necessary to access exclusively when each PE allocates new data structures, even though they will be shared among PEs.

Separate data management and data sharing enables each PE to schedule goal reductions depth-first, i.e., most goal-records and their environments can stay in each separate management area (i.e., in each cache memory). The synchronization overhead is also reduced as in sequential execution. (See Section 3.2.)

#### 2.2.3 Load Balancing

Generally, the separation of the ready-queue makes it difficult to balance the loads and increases the communication to realize a good load balance. We believe that *on-demand distribution* is effective to realize a good balance while reducing the amount of wasteful communication among PEs. We compared the following distribution methods.

- *Rq-i* Method: This method distributes the goals to an idle PE by using a global flag. This flag is set to request a new goal of other busy PEs when the PE becomes idle. By this initiation, a busy PE, which first finds the flag at the *body goal fork*, sends a goal-record to the idle PE instead of enqueuing it in its own ready-queue.
- *Rq-m* Method: This method distributes the goals to an idle PE by using the request message. This request message is sent to a busy PE from the idle PE. The busiest PE is selected by using the global information of each ready-queue length. On receiving the request message, the PE sends the goal-record from its ready-queue to the idle PE.
- *Rand* Method: This method also uses the separated ready-queue. The goal distribution, however, is done at the *body goal fork*, and its destination is decided at random.
- *Comm* Method: This method uses one ready-queue, shared among all PEs. Since each PE enqueues (or dequeues) the goal to (from) this ready-queue, it is not necessary to support the special mechanism for load balancing.

The first two methods are based on on-demand distribution, and the other two methods are introduced for comparison.

### 2.3 Implementation Issues of the KL1 Parallel System

There are two major issues in the implementation of the KL1 parallel system: the processor communication mechanism and the locking mechanism.

#### (1) Processor Communication Mechanism

To realize the above goal distribution methods, it is necessary to implement a PE communication mechanism. In our parallel system, we adopt a message-based communication mechanism in addition to communications using the shared variable cells.

This message-based communication mechanism is viewed as follows. First we set up a post, which receives the messages from other PEs. A PE which wants to send the message links the message to the destination post. Then each PE checks its post for receiving the messages at each reduction. There are three kinds of message: goal-request, goal-send, and messages controling the start and end of the KL1 system.

#### (2) Locking Mechanism

In the Balance 21000 [13], exclusive data access is realized by the lock operations which use a *Test-and-Set memory* called the *Atomic Lock Memory (ALM)*. This direct approach may place an unnecessary burden on the bus and has a restriction on available lock numbers. Because of these, the Balance system provides a "soft" lock, called a *shadow* lock, to use a copy of the lock in shared memory instead of accessing the ALM directly. To use the "soft" lock, our data object is constructed of 8 bytes, i.e., 1 byte for the lock, 2 bytes for the data type tag, 4 bytes for the data value, and 1 unused byte.

The lock operations of the KL1 parallel system are classified as follows:

H-lock is used for exclusive data access in the shared variable cells. Because of the single-assignment feature of KL1, the shared variables can be instantiated by *body unification*. However, by adopting the *bind-hook* mechanism in the suspend operation, the shared variables can be rewritten at the suspend operation. Therefore, the lock operations on the shared variable cells are limited to instantiations of shared variables and suspend operations.

S-lock handles messages in processor communications. The message link operation is done with S-lock.

M-lock is used for maintenance of the process termination. This maintenance is done by the children counter of each processor and the global counter. The children counter of each processor is used by each processor locally, so exclusive access is not necessary for each children counter. However, it is necessary for the global counter. M-lock is used in the global counter access.

## 3 Characteristics of Benchmark Programs

### 3.1 Parallelism

To discuss the system's ability to explore parallelism, we must know the potential parallelism of each benchmark program. Here, we define the two kinds of parallelism under the constraint that the basic unit of parallel processing in our KL1 parallel system is one goal reduction. To define these types of parallelism, we regard the execution of the program as a reduction tree. The reduction tree can be expressed as: the top node is a query goal and each branch node is recursively created by a reduction.

The first parallelism is the *average breadth* of the reduction tree as defined below.

$$average\ breadth = \frac{total\ number\ of\ nodes}{maximum\ depth\ of\ reduction\ tree}$$

This parallelism can be measured by sequential systems which employ breadth-first scheduling. Here, breadth-first scheduling schedules all parallel goals for each goal reduction at one time. The *average breadth* shows the average number of PEs required in the computation if there is an infinite number of PEs.

The other type of parallelism is the *distributable goal ratio*, assuming depth-first scheduling. Here, depth-first scheduling schedules the leftmost goal repeatedly, then the forked body goals from left to right. Each PE can get the load (goal) from another PE if the other PE has an extra goal in its ready queue. This parallelism can be defined as:

$$distributable\ goal\ ratio = \frac{\sum_{i=0}^{total\ number\ of\ nodes}(branches_i - 1)}{total\ number\ of\ nodes}$$

$Branches_i$ is the number of branches at the $i$-th node. For example, the *append* program creates only one goal in each reduction, so there is no parallelism.

## 3.2 Relationship among Parallel Goals

In KL1 execution, parallel goals can be synchronized by the instantiation of shared variables, so that if a goal was scheduled before instantiating the necessary variables, that goal has to wait as a suspended goal until the instantiation of those variables. If the system can schedule perfectly parallel goals for a given program, there will be no overhead for the suspension. If not, the system must pay a tremendous price for synchronization among parallel goals. However, it is almost impossible to schedule perfectly in parallel execution. On the other hand, some kinds of program can be executed without suspension by simply using depth-first scheduling in sequential execution. This is because repeated invocation of the leftmost goal can be regarded as a process and such processes have one-directional relations, with an exact from-left-to-right order of body goals.

## 3.3 Benchmark Programs and Their Basic Characteristics

The benchmark programs used in the evaluation of KL1 parallel system are:

### (1) Three 8-queen Programs:

The 8-queen program is a famous problem that searches all solutions. The first program (8q-m) is translated from an OR-parallel Prolog program. Solutions by each search process are gathered by stream merge procedures. The second (8q-s) is a highly optimized program which performs no suspensions. The third (8q-l) is a program which uses the layered stream algorithm [11].

### (2) Quicksort Program (qsrt):

A list of 512 numbers is sorted in this program, forking sort processes as a binary tree.

Table 1: Parallelism of Benchmarks

| | 8q-m | 8q-s | 8q-l | BUP | qsrt | prim | mxf |
|---|---|---|---|---|---|---|---|
| Reductions | 108K | 39K | 19K | 36K | 8K | 17K | 40K |
| Average breadth | 647 | 563 | 511 | — | 14 | 15 | 44 |
| Distributable goal ratio | 0.76 | 0.40 | 0.64 | 0.74 | 0.16 | 0.01 | 0.11 |
| S/R (depth-first) | 0 | 0 | 0 | 0 | 0 | 0 | 0.40 |
| S/R (breadth-first) | 0.28 | 0 | 0.13 | — | 0.37 | 0.88 | 0.40 |

(3) Prime Number Generator Program (prim):

This program is a simple *generate-and-test* type program.

(4) Bottom Up Parser (BUP) Parsing Program:

This program searches all alternative parsing trees of Japanese-language sentences using the BUP algorithm [10]. Like *8q-m*, solutions are gathered by stream merge procedures.

(5) Maximum Flow Program (mxf):

Nodes and links in a given network are represented by KL1 goals and message streams among them. The flow in each link is restricted. This program finds the maximum flow by sending and receiving messages between neighboring goals.

Table 1 shows the basic characteristics of the above benchmarks. *Reductions* means the total number of reductions, and the *average breadth*[1] and the *distributable goal ratio* are the potential parallelism for each benchmark. *S/R* is the number of suspensions per reduction of each benchmark when one processor executes by both depth-first and breadth-first scheduling.

Table 1 indicates that the three kinds of 8-queen program and *BUP* have both kinds of parallelism. On the other hand, *qsrt*, *prim* and *mxf* do not have much parallelism. Table 1 also indicates that the number of suspensions are low (zero) in depth-first scheduling for most programs. In other words, there is a relationship among goals which can be treated by simple scheduling, i.e. depth-first scheduling.

## 4 Evaluation of the KL1 Parallel System

This section evaluates the results gathered by the KL1 parallel system for the following reasons.

- To find the effectiveness of *on-demand distribution* and the separation of ready-queues in KL1 parallel execution
- To verify that substantial speedup can be attained from the benchmarks on the KL1 parallel system
- To understand the factors which reduce the speedup

[1]The *average breadth* statistics of BUP could not be calculated because the breadth-first scheduling system does not support the *otherwise* predicate.

Table 2: Sequential Performance

| | 8q-m | 8q-s | 8q-l | BUP | qsrt | prim | mxf |
|---|---|---|---|---|---|---|---|
| Execution time (sec) | 125 | 54 | 59 | 52 | 10 | 21 | 147 |
| RPS | 974 | 771 | 369 | 729 | 800 | 790 | 271 |
| KL1B/red | 10.5 | 14.4 | 32.6 | 15.0 | 14.0 | 14.0 | 38.8 |
| KL1B/sec | 10K | 11K | 12K | 11K | 11K | 11K | 11K |

Table 3: Suspensions in Load Balancing (BUP)

| | *comm* | *rand* | *rq-i* | *rq-m* |
|---|---|---|---|---|
| Work rate | 99% | 94% | 96% | 96% |
| Suspensions | 9.3K | 6.2K | 1.0K | 0.5K |
| KL1B ratio (PE=8/PE=1) | 1.27 | 1.15 | 0.98 | 0.98 |

## 4.1 Sequential Performance

This subsection gives the performance and several basic statistics of the KL1 parallel system running on one PE[2]. Table 2 shows the execution time (sec), reductions per second (RPS), KL1-B instructions per reduction (KL1B/red), and KL1-B instructions per second (KL1B/sec) for each benchmark. The performance of the *append* program is 1,400 RPS. The basic statistics are as follows:

- Average execution time of a KL1-B instruction: about 90 $\mu$sec
- Lock operation time: about 30 $\mu$sec
- Retry time in a lock contention: about 5 $\mu$sec
- Manipulation time of a message chain: about 230 $\mu$sec
- Average time of a message analysis: about 300 $\mu$sec
- Average time of a suspension *and* a resumption: about 500 $\mu$sec

## 4.2 Comparison of Load Balance Methods

Figure 2 shows the speedup ratios of the BUP program which are obtained by changing the load balance methods. (See Section 2.2.3.) It clearly shows that the *on-demand distribution* methods (*rq-i* and *rq-m*) can obtain better performance than the other methods (*comm* and *rand*).

To find the factors which reduce the performance, we gathered more detailed statistics using eight PEs. The results indicate that the biggest factor is the increase in suspensions. Table 3 shows the *work rate*, the number of suspensions, and the ratio of the number of KL1B instructions on eight PEs to that of one PE. *Work rate* is defined as follows:

$$\textit{work rate} = 1 - \frac{\textit{average PE's busy waiting time over all PEs}}{\textit{average execution time over all PEs}}$$

Each distribution method has almost the same *work rate*, but the number of executed instructions is very different. The additional instructions on the eight-PE system are caused by the increased number of suspensions.

[2]The Balance system uses the NS32032 as a CPU, and its performance is 0.7 MIPS [14].

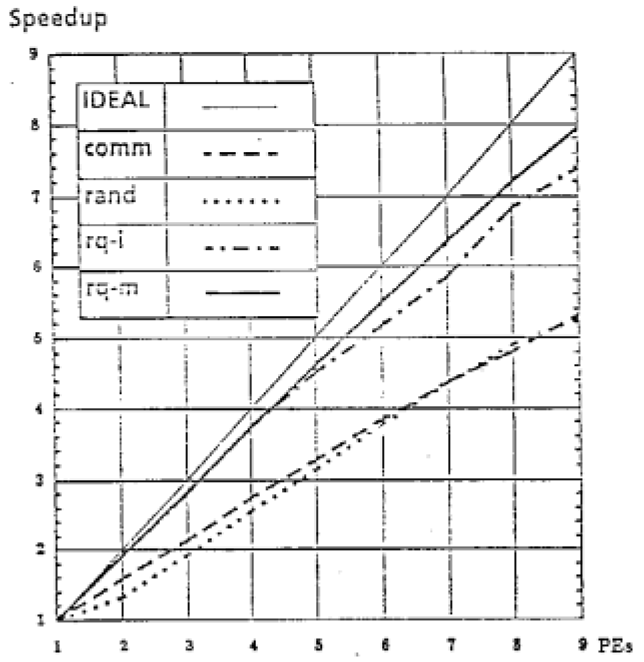

Figure 2: Comparison of Load Balance Methods (BUP)

As shown in Figure 2 and Table 3, the performance of *rq-m* is only slightly better than *rq-i*. The *rq-i* distributes goals only at the body goal fork, and the *rq-m* distributes goals from a ready queue. Although more evaluation is necessary, we expect the characteristics of distributed goals to cause a difference in performance.

## 4.3 Speedup

Figure 3 shows the speedup of each benchmark under the *rq-m* distribution method. This indicates that the speedup is related to the parallelism which is defined in Section 3.1. That is, if the benchmark has enough parallelism, it can obtain high speedup on our KL1 parallel system. More detailed analyses are given in the following subsections.

## 4.4 Analyses of Degrading Speedup

There are four factors which may degrade the speedup:

1. Increase of idle time because of inefficient load balance (IDLE)
2. Overhead of lock operations (LOCK)
3. Overhead of inter-PE communications (COMM)
4. Increase in computations because of suspensions (SUSP)

Figure 4 shows the proportions of each factor and real efficiency. These statistics are gathered by using eight PEs under the *rq-m* distribution method. The proportions are based on the ideal performance, where one PE's performance is multiplied by the number of PEs used (eight). The performance of one PE is not degraded by the above factors. The real efficiency is the ratio of the real performance using eight PEs and the ideal performance. Each factor is calculated from the numbers and the time in one operation.

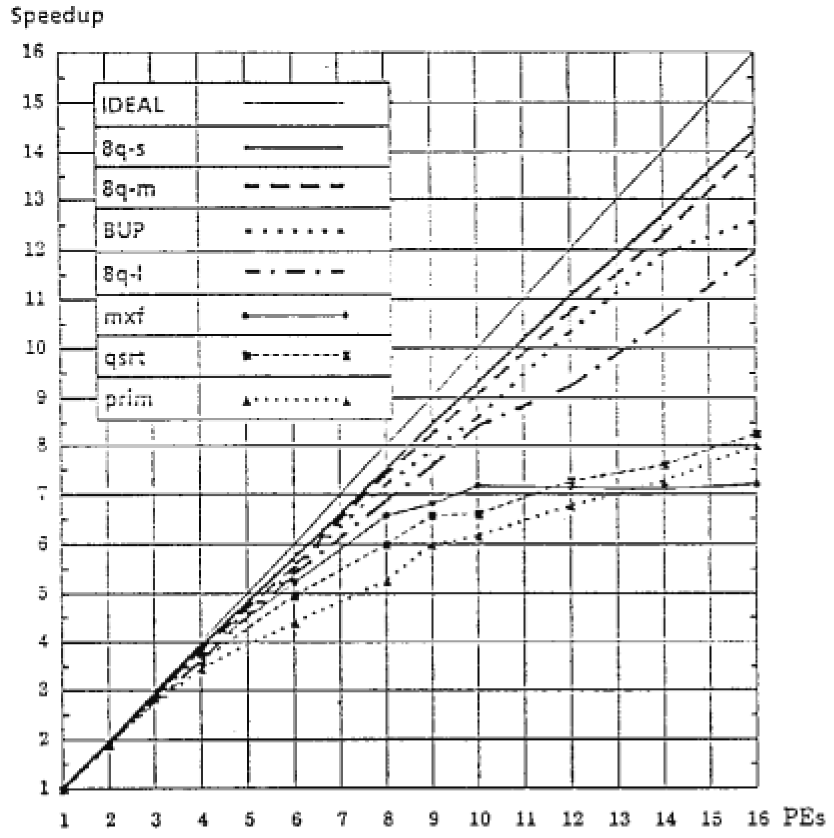

Figure 3: Speedup

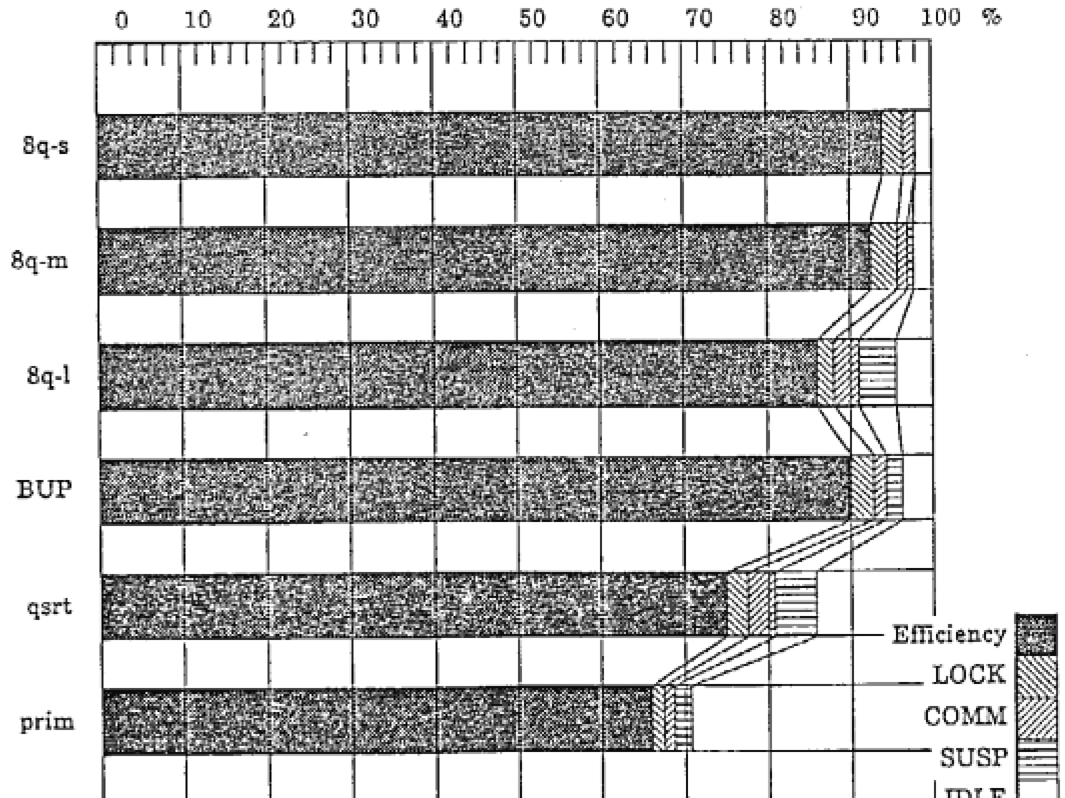

Figure 4: Proportion of the Degradation in Speedup

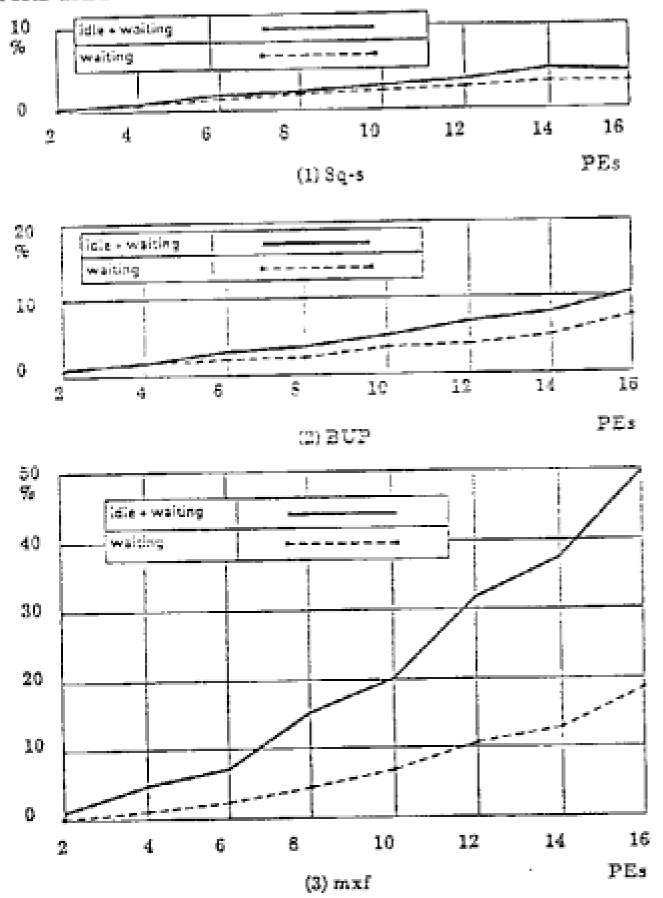

Figure 5: Proportion of Idle Times

Figure 4 indicates that the main factor in deciding real efficiency is idle time. The next is the increase of executed instructions by suspensions, although it depends on the benchmark. The overhead of lock operations and inter-PE communications are minimal, only 1% to 5% each. This is an almost constant overhead for each benchmark.

### (1) Increase in Idle Time

The idle time is divided into two kinds of time. The first is *real idle time*, where all the other PEs do not have distributable goals. The second is the *waiting time* from the request to the answer. Figure 5 shows the percentage of idle time in the work time, changing the number of PEs under the *rq-m* distribution method. The benchmarks used are *8q-s*, *BUP* and *mxf*, because they differ in speedup. (See Figure 3.) In this figure, the solid line shows the sum of the *real idle time* and the *waiting time*, and the broken line shows the *waiting time.*

Figure 5 indicates that the main factor of increase in idle time is the increase in *real idle time.* The increase in *real idle time* comes from the small parallelism of its benchmark, i.e., in 8q-s and BUP which have enough parallelism, the *real idle time* is very small. In mxf, which dose not have enough parallelism, however, the *real idle time* is very large.

The increase in *waiting time* is caused by scrambling to get a goal for many idle PEs, i.e., other PEs which could not get the goal must wait for a certain period then try to get a goal again, so that if there are not enough goals to distribute, an idle PE must wait a long time.

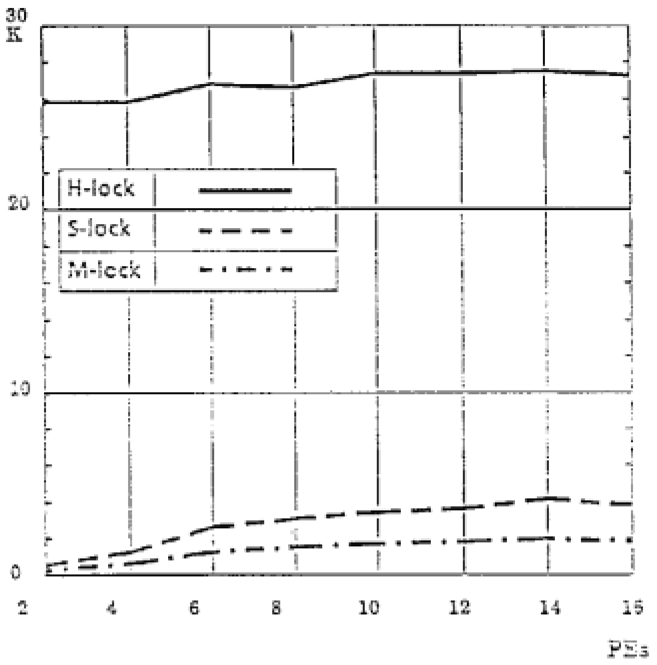

Figure 6: Number of Lock Operations

Table 4: Goal Distribution Ratio

| Distribution methods | rand | rq-m | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Benchmarks | BUP | BUP | 8q-m | 8q-s | 8q-l | qsrt | prim | mxf |
| Distribution/red | 44.7% | 1.8% | 2.0% | 1.7% | 6.7% | 2.5% | 1.0% | 5.3% |

## (2) Overhead of Lock Manipulation

The overhead of the lock manipulation is very small. (See Figure 4.) However, the features of lock manipulations are important, so we gathered two kinds of statistics, the number of lock operations (see Figure 6) and the lock contention ratio (see Figure 7), for three kinds of locks: the **H-lock**, S-lock, and M-lock. (See Section 2.3.) These statistics are gathered by using the *8q-s* benchmark and changing the number of PEs.

Figure 6 indicates that the number of lock operations increases only slightly according to the number of PEs. Figure 7 indicates that the lock contentions of the **H-lock** are negligible. The reason for the small lock contention of the **H-lock** comes from the KL1 characteristics [1], i.e., there are few multiple references to the same data object. On the other hand, the increase in the **S-lock's** and the **M-lock's** lock contentions are rather large, because exclusive access using them concentrates on only a few global data. In this emulator, which was implemented on the Balance system, this lock contention is still small. However, this indicates that it is necessary to estimate the **S-lock's** and M-lock's lock contentions in the design of the PIM hardware.

## (3) Communications Overhead

The communications overhead is small (see Figure 4), mainly because *on-demand* distribution is adopted. number of the distributed goals and that of the executed goals. This is called the *distribution ratio*. This statistic is gathered by using eight PEs. As stated in Section 4.1, the communication cost is large, almost half that of one goal reduction. However, the amount of communication is small because on-demand distribution is adopted. (See Table 4.)

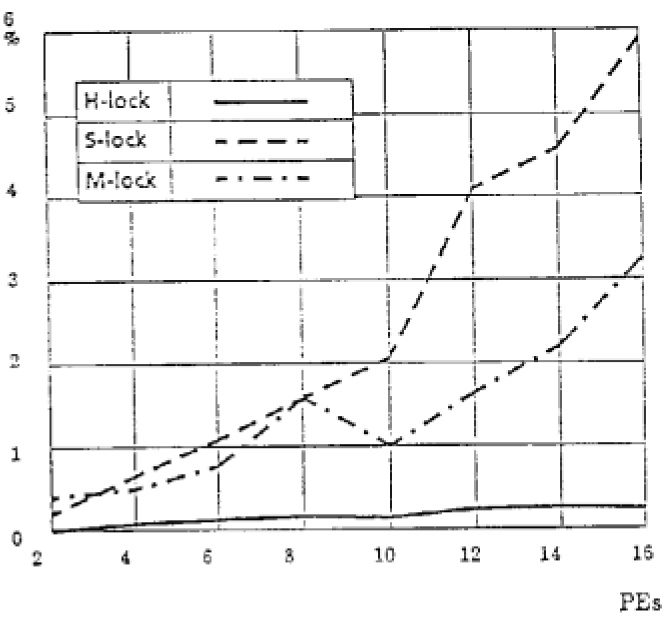

Figure 7: Lock Contention Ratio

Table 5: Statistics of Suspensions

| | 8q-m | 8q-s | 8q-l | BUP | qsrt | prim | mxf |
|---|---|---|---|---|---|---|---|
| Suspension/red *rq-m* | 0.01 | 0 | 0.10 | 0.02 | 0.08 | 0.04 | 0.42 |
| Suppression ratio | 3% | 0 | 10% | - | 23% | 4% | 95% |

## (4) Increase in Computations Because of Suspension

The increase of computations due to suspension affects speedup. The reason for the increase in computations due to suspension is that suspension requires an extra *suspend operation*, described in Section 2.1, and extra instructions to retry the *passive-part* executions. Although these extra instructions depend on the number of clauses, there is an average of about 10 extra instructions over benchmark programs[3]. Therefore, the suspension cost is very large if the suspension has occurred.

However, as shown in Figure 4, the reduction in performance caused by suspension is not large in our KL1 parallel system with on-demand goal distribution, *rq-m*. There are two reasons for low suspension overhead: on-demand distribution and depth-first scheduling. Section 4.2 showed that on-demand distribution can suppress suspensions. Table 5 shows the suspension statistics. *Suspension/red* is the number of suspensions per reduction in *rq-m*. *Suppression ratio* is defined as follows:

$$Suppression\ ratio = \frac{rq\!-\!m(suspensions)}{bf(suspensions)}$$

*bf(suspensions)* is the number of suspensions under breadth-first scheduling by one processor and *rq-m(suspensions)* is the number under the *rq-m* distribution method. *Suppression ratio* shows how much the depth-first scheduling with on-demand distribution suppresses suspensions.

Table 5 shows that depth-first scheduling suppresses suspensions in most benchmarks, except for *mxf*. Depth-first scheduling in not effective for *mxf* because *mxf* requires frequent handshaking among parallel KL1 goals. It is necessary to study more effective scheduling for programs like *mxf*.

[3]These analyses were done by a system in which the indexing instructions have not implemented.

## 5 Conclusions

This paper presented the design decisions for the KL1 parallel system, which is implemented on the genuine multiprocessor, Balance 21000. It also gave the characteristics of benchmarks, which are defined by the sequential system, and the first detailed results of some benchmarks which were gathered by the KL1 parallel system.

Our results indicate that our strategies, *on-demand* distribution and the separation of ready-queues, are efficient for genuine multiprocessors. Substantial speedup can be obtained from several benchmarks according to their potential parallelism which can be easily defined by using the sequential system.

More detailed analyses of the factors reducing speedup indicate that the main factor is the low *work rate* which comes from the benchmark's poor potential parallelism. The other overhead, the lock manipulation for the exclusive data access in parallel and the communications for the realization of the good load balance, is very small. The overhead from the increase of suspensions by parallel execution, however, is larger than we expected. To keep the increase of suspension to a minimum, we introduce a simple scheduling strategy, depth-first scheduling in each PE. The depth-first scheduling is effective for most benchmarks.

## Acknowledgment

We extend our thanks to the ICOT Director, Dr. Kazuhiro Fuchi, and the chief of the fourth research laboratory, Dr. Shunichi Uchida, who gave us the opportunity to pursue this research. We also appreciate the help of our colleagues in ICOT, especially Mr. Toshihiko Miyazaki for letting us use the breadth-first scheduling system and Dr. Evan Tick for valuable discussions.

## References

[1] T. Chikayama and Y. Kimura. Multiple Reference Management in Flat GHC. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 276–293, 1987. Also published as ICOT Technical Report TR-248.

[2] T. Disz, E. Lusk, and R. Overbeek. Experiments with OR-Parallel Logic Programs. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 576–600, 1987.

[3] I.T. Foster. Logic operating sytems: design issues. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 910–926, 1987.

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

[5] B. Hausman, A. Ciepielewski, and S. Haridi. OR-Parallel Prolog Made Efficient on Shared Memory Multiprocessors. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 69–79, 1987.

[6] M. V. Hermenegildo. Relating Goal Scheduling, Precedence, and Memory Management in AND-parallel Execution of Logic Programs. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 556–575, 1987.

[7] N. Ichiyoshi, T. Miyazaki, and K. Taki. A Distributed Implementation of Flat GHC on the Multi-PSI. In *Proceedings of the Fourth International Conference on Logic Programming*, 1987. Also published as ICOT Technical Report TR-230.

[8] Y. Kimura and T. Chikayama. An Abstract KL1 Machine and Its Instruction Set. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 468–477, 1987. Also published as ICOT Technical Report TR-246.

[9] A. Matsumoto, M. Sato, et al. *Locally Parallel Cache Designed Based on KL1 Memory Access Characteristics.* TR 327, ICOT, 1987. Also submitted to ISCA 1988.

[10] Y. Matsumoto. BUP: A Bottom-up Parser Embedded in Prolog. *New Generation Computing, OHMSHA Ltd. and Springer-Verlag*, 1(2):145–158, 1983.

[11] A. Okumura and Y. Matsumoto. Parallel Programming with Layered Streams. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 224–232, 1987.

[12] M. Sato, A. Goto, et al. KL1 Execution Model for PIM Cluster with Shared Memory. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 338–355, 1987. Also published as ICOT Technical Report TR-250.

[13] Inc. Sequent Computer Systems. *Balance 8000/21000 Guide to Parallel Programming.*

[14] Inc. Sequent Computer Systems. *Balance 8000/21000 Technical Summary.*

[15] E.Y. Shapiro. *A Subset of Concurrent Prolog and Its Interpreter.* TR 003, ICOT, 1983.

[16] A. Takeuchi and K. Furukawa. Bounded Buffer Communication in Concurrent Prolog. *New Generation Computing, OHMSHA Ltd. and Springer-Verlag*, 3(4):359–384, 1985.

[17] K. Taki. The Parallel Software Research and Development Tool : Multi-PSI system. In *France-Japan Artificial Intelligence and Computer Science Symposium 86*, pages 365–381, October 1986.

[18] S. Taylor, A. Safra, and E. Shapiro. A Parallel Implementation of Flat Concurrent Prolog. *International Journal of Parallel Programming*, 15(3):245–275, 1986.

[19] P. Tinker and G. Lindstrom. A Performance-Oriented Design for OR-Parallel Logic Programming. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 601–615, 1987.

[20] K. Ueda. *Guarded Horn Clauses.* TR 103, ICOT, 1985.

[21] K. Ueda. *Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard.* TR 208, ICOT, 1986. (Also to appear in Programming of Future Generation Computers, North-Holland, Amsterdam, 1987.).

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

[23] D.H.D. Warren. The SRI Model for OR-Parallel Execution of Prolog. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 92–102, 1987.