An Overview of The System Software of A Parallel Relational Database Machine GRACE

Shinya FUSHIMI*†, Masaru Kitsuregawa**, and Hidehiko Tanaka*

*Department of Electrical Engineering,
The University of Tokyo, Tokyo, Japan

**Institute of Industrial Science,
The University of Tokyo, Tokyo, Japan

ABSTRACT

This paper outlines the system software of a parallel relational database machine GRACE, and describes its execution and control of relational operations based on the *data stream oriented processing*. The system software is organized in a hierarchy, and the execution of a relational operation and its operand data are encapsulated and controlled in the form of *task*. The *data stream control protocol* between modules in a task makes tasks autonomous objects. The system software we propose eliminates the greater part of possible control overheads first by adopting the task-level granularity for the execution and control, then by executing the operation along the flow of operand data. The former reduces the control overhead for enabling the execution of a relational operation, while the latter hides the execution behind the I/O's or data transfer. Its preliminary implementation on the software simulator of GRACE is also reported. In addition, the novel virtual space management algorithm is proposed, which enables us to handle a large data stream quite efficiently.

1. Introduction

The past decade has seen considerable efforts directed towards exploring parallel architectures for relational database processing [Ozka75, DeWi78, Kits82, Kim84]. These research efforts have clarified two substantial performance factors of database machines: the *control and data transfer overhead* for parallel execution of relational operations, and the classical *I/O bottleneck problem* [DeWi86]. The typical example which convinces us of these points is the DIRECT prototype implementation [Bora82]. The designers of DIRECT reported that, although relational operations were executed with sufficient parallelism, the performance was not improved as expected due to the heavy control and data transfer overhead and the rather low I/O performance (in spite of dataflow control and parallel disk I/O mechanism of DIRECT).

As for the I/O bottleneck problem, we have already developed the adaptive multidimensional clustering algorithm [Fush85], and the highly functional disk system [Kits86], both of which have shown to be able to achieve the drastic performance improvement. For example, our functional disk system can execute Wisconsin benchmark including join in 6.2 seconds, compared with 10.2 minutes for Ingres and 1.8 minutes for its commercial version. With such background, our major concern is now the efficient system software to control the complexed query processing in the parallel relational database machine.

Here we examine the control and execution strategy of DIRECT. DIRECT adopted the page-level granularity for enabling processors to execute relational operations [Bora80]. That is, operand relations are partitioned into a set of pages, which are processed by multiple processors in parallel. The page-level granularity enables the machine to incorporate the data flow control where a processor can be fired when at least one operand page is formed. It seems that the page-level granularity is the best choice to execute relational operations in the multiprocessor architecture. We notice, however, that every activation of processors should be preceded and followed by communications with the controller and data transfers to/from memory modules (Figure 1); to start an operation to a page, a processor should first obtain the page address from the controller, then read data

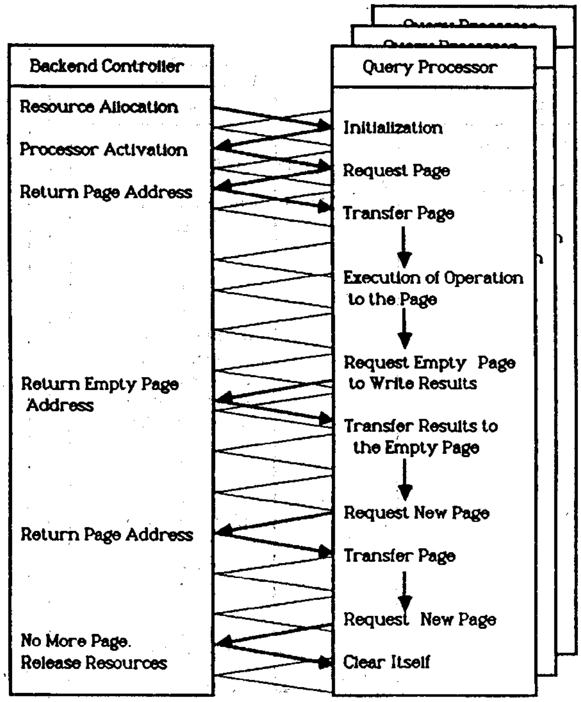


Figure 1. Execution of Relational Operation in DIRECT

[†] Currently with Mitsubishi Electric Co., Kamakura City, Japan

in the page from a memory module into its local memory. Similarly, to wind up the operation, a processor should obtain an empty page and write the results in it. Since data transfer consumes rather long time, and the controller becomes too busy to provide these page addresses to multiple processors, more than half of the total processing time was consumed by such overheads[Bora82]. Note that the performance cannot be improved by employing more processors, or by making processors faster. It can be improved to our satisfaction only when these overheads are removed by the efficient control mechanism, or the *system software*.

Observations above imply that relational operations should be executed by much coarser granularity at least to reduce the control overhead, and that the execution of relational operations should be overlapped with data transfer. The *data stream oriented processing* can meet these requirements (Figure 2). In the data stream oriented processing, the unit of execution and control is a whole set of tuples referred to by the operation. Thus, the controller of the machine is invoked much less frequently than in DIRECT: it is called only when an operation starts or finishes. Also, all of operations are executed *keeping up with the flow of data* (hence the name data stream oriented processing). In other words, all of operations are hidden behind I/O's or data transfers.

A parallel relational database machine GRACE [Kits84] fully achieves the data stream oriented processing. In GRACE, join is executed in $O((N+M)/m)$ time by the novel parallel algorithm based on hash and sort [Kits83], where N and M are sizes of operand relations, and m is the number of *sources* which generate data streams in parallel. In this paper, we present its system software to control the query processing, and describe how the data stream oriented processing is achieved in the machine.

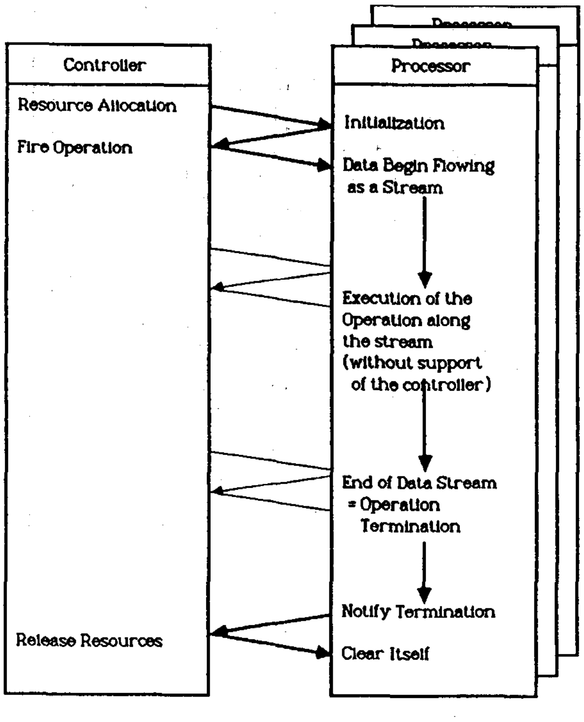

Processor
Processor
Controller
Processor
Resource Allocation
Fire Operation
Release Resources
Initialization
Data Begin Flowing as a Stream
Execution of the Operation along the stream (without support of the controller)
End of Data Stream = Operation Termination
Notify Termination
Clear Itself

Figure 2. Execution of Relational Operation Based on Data Stream Oriented Processing

In GRACE, every relational operation is executed in the form of *task*. Task is the encapsulation of the execution of relational operation and its operand data stream, and generally consists of *source space*, *filtering*, *clustering*, and *sink space*, along with a *data stream* to be processed. The system software controls the complexed query processing by the units of task, while a task can run quite autonomously by the *data stream control protocol* between the modules involved in it. This task-level granularity and the data stream control protocol distinguish our system software from others.

Although our system software design includes the concurrency control and transaction management, we omitted their detailed description because of the space limitation. Also, the formal model used for the design is omitted here. The details on them can be found in [Fush86].

The remainder of the paper is organized as follows. In section 2, the architectural description of GRACE and its query processing algorithms are given. In this section, the novel *virtual space management algorithm* is also proposed. The overall organization of its system software is presented in section 3. The system software is organized in a hierarchy of four layers: *primitive algorithms layer*, *intra-task control layer*, *inter-task control layer*, and *transaction management layer*. Section 4 and 5 outline the design and organization of the intra- and inter-task layers. Section 6 describes the preliminary implementation of the system software on the software simulator of GRACE. The conclusion is given in section 7, along with the brief description of the concurrency control and transaction management, and the current status of the GRACE project.

2. A Parallel Relational Database Machine GRACE

2.1. Hardware Architecture of GRACE

GRACE is a parallel relational database machine currently being developed at The University of Tokyo [Kits82, Kits84]. GRACE aims at achieving high performance even for the join-intensive applications by incorporating the data stream oriented processing. The linear time parallel join algorithm based on the clustering property of *hashing* and the linear time *sorting* supports the data stream oriented processing in GRACE.

The global architecture of GRACE is shown in Figure 3. GRACE consists of four kinds of fundamental modules: *processing module*, *memory module*, *disk module*, and *control module*. Processing modules are responsible for the execution of relational operations. Memory modules offer the temporary, or *staging storage* for data streams. The database itself is stored in disk modules. Control modules are in charge of the control of the machine. These modules are connected each other through two ring buses, *processing ring* and *staging ring*. These rings are so called time-division, multiple channel ring bus in which multiple channels are activated and available for the communication between modules. Rings are implemented by shift registers in *ring bus interface units* connected in a circle.

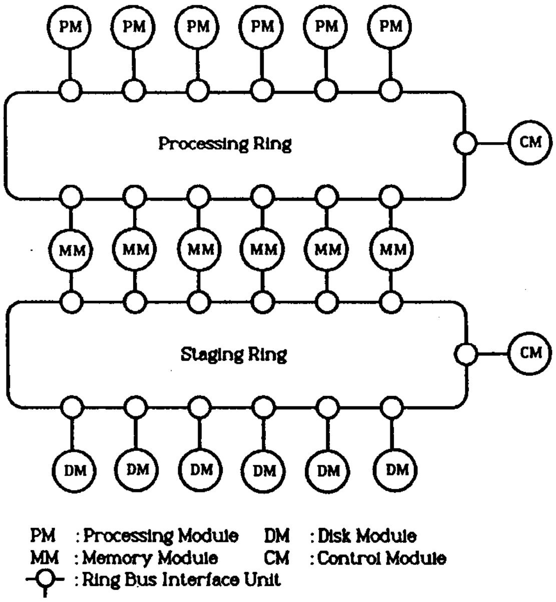

PM
Processing Ring
CM
MM
Staging Ring
CM
DM
PM : Processing Module DM : Disk Module
MM : Memory Module CM : Control Module
: Ring Bus Interface Unit

Figure 3. Hardware Organization of GRACE

2.2. Query Processing in GRACE

GRACE implements the novel query processing algorithm based on hash and sort [Kits83]. Consider the query consisting of two selections to relations R and S followed by a join of selected results. To execute this query, first two selections to R and S are activated simultaneously. Data streams are generated from disk modules storing R and S and directed to the "sink", a set of memory modules (Figure 4). The query processing is now said to be in the *staging phase*. Let's take a close look at this phase. In the staging phase, tuples emerging from disks are first directed to the *filter processor* in the disk module, and applied the selection operation. This processor also eliminates unnecessary attributes from tuples. Only tuples which passed this test are then sent to the *hashing unit* by which the hash value of the join attribute is attached to each tuple. Tuples having the same hash value form one *cluster* [1]. Note that tuples in one cluster cannot be joined with those in other clusters. The net effect is that join is decomposed into small joins which can be processed independently (Figure 5). These tuples are finally sent to sink memory modules allocated to this data stream. As shown in Figure 4, each cluster is uniformly distributed over sink memory modules so that variable sizes of clusters generated by hashing do not affect the memory utilization [Kits83].

It should be mentioned that the so-called joinability filter can be incorporated in disk modules by using filter and hashing processors. Although both of joinability filter and our clustering technique utilize hashing, they are in principle different techniques, and can coexist in GRACE [Kits83].

When the processing completes the staging phase, it then goes to *processing phase*, in which the subsequent join operation is performed (Figure 6). In this phase, memory

[1] A cluster is also called a *bucket*.

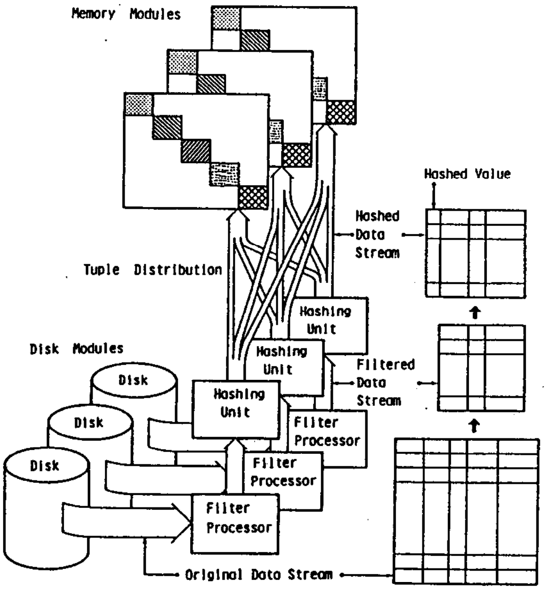
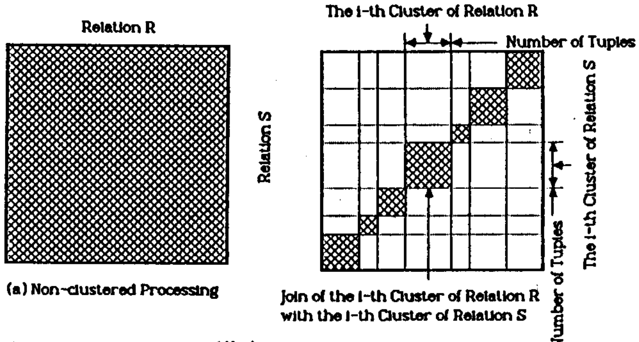


Figure 4. An Overview of Staging Phase


Relation R
Relation S
The i-th Cluster of Relation R
Number of Tuples
Number of Tuples
The i-th Cluster of Relation S
Join of the i-th Cluster of Relation R with the i-th Cluster of Relation S

(a) Non-clustered Processing

Figure 5. Clustering Effect of Hash

(b) Clustered Processing

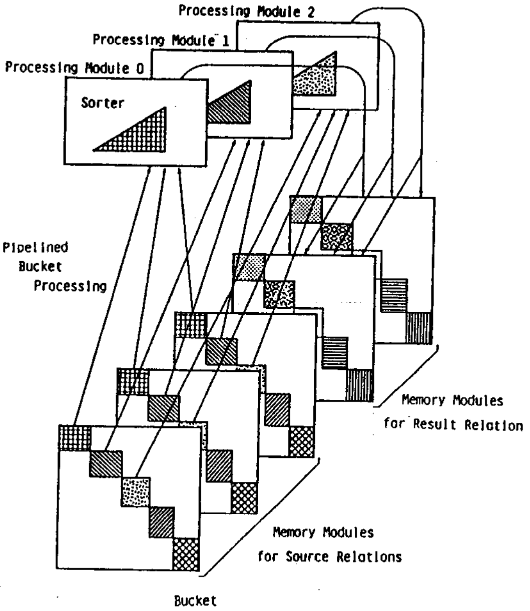
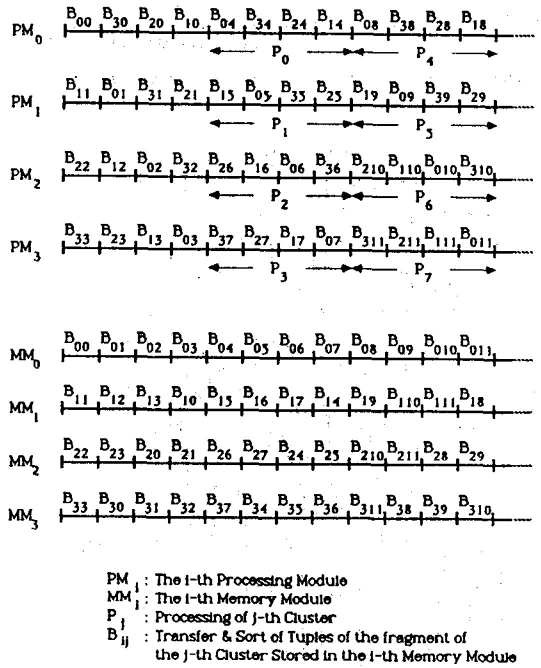


Figure 6. An Overview of Processing Phase

modules which absorbed tuples resulted in preceding selections are switched to the "source" of a data stream. The same number of processing modules as source memory modules are activated to perform join operation on the cluster by cluster basis. They work in parallel, or more precisely, in a pipelined fashion (Figure 7). When activated, the *i*-th processing module first visits the *i*-th memory module to obtain the fragment of the *i*-th cluster stored therein. If the fragment is consumed, the *i*-th processing module changes its partner to the (i-1 modulo n)-th memory module to get the remaining fragments of the *i*-th cluster, where n is the number of source memory modules. The whole cluster is collected when the processing module visited all of memory modules. While collecting the cluster fragments, processing modules conduct the join operation to the incoming streams of fragments. In a processing module, the linear time algorithm using *pipelined merge sorter* is applied along the flow of incoming data. The pipelined merge sorter can sort a data stream of tuples of variable lengths at 3MB/sec [Kits85]. The sorted stream is directed to the *tuple manipulation unit* in the processing module, in which the intra-cluster join is executed. The algorithm in it can be quite simple because data are already sorted. Since the sorting and its subsequent operations are performed along the flow of incoming stream, a processing module becomes ready to process the next cluster just when it finished gathering the current cluster.

As processing modules execute the join operation, the result tuples are gradually formed. The *hashing unit* located in the processing module is used to partition the result tuples suitable for the next operation. The final output from the processing modules are forwarded to another set of memory modules, or the "sink". Result tuples are stored in



PM$_0$
B$_{00}$ B$_{30}$ B$_{20}$ B$_{10}$ B$_{04}$ B$_{34}$ B$_{24}$ B$_{14}$ B$_{08}$ B$_{38}$ B$_{28}$ B$_{18}$
P$_0$
P$_4$
PM$_1$
B$_{11}$ B$_{01}$ B$_{31}$ B$_{21}$ B$_{15}$ B$_{05}$ B$_{35}$ B$_{25}$ B$_{19}$ B$_{09}$ B$_{39}$ B$_{29}$
P$_1$
P$_5$
PM$_2$
B$_{22}$ B$_{12}$ B$_{02}$ B$_{32}$ B$_{26}$ B$_{16}$ B$_{06}$ B$_{36}$ B$_{210}$ B$_{110}$ B$_{010}$ B$_{310}$
P$_2$
P$_6$
PM$_3$
B$_{33}$ B$_{23}$ B$_{13}$ B$_{03}$ B$_{37}$ B$_{27}$ B$_{17}$ B$_{07}$ B$_{311}$ B$_{211}$ B$_{111}$ B$_{011}$
P$_3$
P$_7$
MM$_0$
B$_{00}$ B$_{01}$ B$_{02}$ B$_{03}$ B$_{04}$ B$_{05}$ B$_{06}$ B$_{07}$ B$_{08}$ B$_{09}$ B$_{010}$ B$_{011}$
MM$_1$
B$_{11}$ B$_{12}$ B$_{13}$ B$_{10}$ B$_{15}$ B$_{16}$ B$_{17}$ B$_{14}$ B$_{19}$ B$_{110}$ B$_{111}$ B$_{18}$
MM$_2$
B$_{22}$ B$_{23}$ B$_{20}$ B$_{21}$ B$_{26}$ B$_{27}$ B$_{24}$ B$_{25}$ B$_{210}$ B$_{211}$ B$_{28}$ B$_{29}$
MM$_3$
B$_{33}$ B$_{30}$ B$_{31}$ B$_{32}$ B$_{37}$ B$_{34}$ B$_{35}$ B$_{36}$ B$_{311}$ B$_{38}$ B$_{39}$ B$_{310}$

PM$_i$: The i-th Processing Module
MM$_i$: The i-th Memory Module
P$_j$: Processing of j-th Cluster
B$_{ij}$: Transfer & Sort of Tuples of the fragment of the j-th Cluster Stored in the i-th Memory Module

Figure 7. Data Stream Pipelining in Processing Phase

these memory modules in quite the same way as in the staging phase.

Note that this algorithm works well only if clusters are of the same size. Otherwise, a processing module must wait when the memory module which it wants to access next is still being visited by other processing module. As a result, the pipelining is disturbed. The idea to tackle this problem is the *cluster size tuning* which merges the set of small clusters produced by clustering into some number of *processing clusters* whose sizes are nearly equal to the capacity of processing module.

Note also that the preprocessing of data (hashing) is completely hidden behind the execution of the preceding operation. Therefore, no overhead appears even when a query to be processed include many relational operations.

2.3. Task

The processing of a query in general requires it to be decomposed into a number of primitive database operations. In GRACE, these operations include all of relational algebra operations (joins, selections, projections, divisions, etc.), sorting, and aggregation. The execution of primitive database operations are encapsulated in the form of *task*. High-level query such as SELECT FROM WHERE clause in SQL [Cham76] is configured as the tree of tasks called *task tree*.

Intuitively, task consists of "river" and "water". "Water" is a data stream, i.e., flowing tuples of operand relations, while the "river" offers the *execution environment* for the operation. The execution environment will be composed of necessary number of machine resources enough to execute the designated operation. A database operation is executed in the form of task by letting operand data flow through its execution environment being applied the designated operation.

Our design policy is that a task should be so autonomous that it runs without any direction from the controller. If a task can be such a self-contained object, the control module becomes responsible only for preparing and eliminating the execution environment for task, that is, managing resources. Specifically, according to the task tree, the controller appropriately produces "junction"s when it prepares "river"s for data streams. The task tree can be considered to represent how tributary rivers are merged into the main stream.

The execution of primitive database operations which need to access the database directly (selections, etc.), thus executed in the staging phase, is called a *read task*, while that of other operations like joins which are executed in the processing phase are called an *internal task*.

In GRACE, the execution environment of a read task consists of the set of disk modules which store the tuples to be accessed by the corresponding selection, a set of memory modules to store the results, and a set of channels on the staging ring to convey tuples (see Figure 4). Execution environment of an internal task is in turn composed of the set of memory modules which store the operand data, a set of processing modules, a set of memory modules to absorb the results, and a set of channels on the processing ring (Figure 5).

We notice that either of read or internal tasks essentially has the identical structure; either task logically consists of *source space*, which produces the parallel data streams (disks in disk modules in read tasks, or memory modules in internal tasks), *filtering*, which applies several modifications or sieving to data streams (filtering units in disk modules in read tasks, or hardware sorter and tuple manipulation units in processing modules in internal tasks), *clustering*, which partitions a data stream into multiple, independent data substreams (hashing units in disk or processing modules in read and internal tasks respectively) and, *sink space*, which absorbs the result data streams (memory modules in both of tasks). This observation can be formalized as the *data stream model* which can be a general framework for the system software design of data stream oriented relational database machines [Fush86].

2.4. Virtual Space Management Based on Clusters

From the view point of the structure of execution environment, the most crucial problem is the overflow in the sink space. To start a task, enough capacity should be prepared as the sink space; otherwise, if the unexpectedly large stream is resulted, the overflow occurs in the sink space, disabling the processing to proceed further. Since precise estimation of the size of the result data stream cannot be expected, some means must be taken.

Our solution to this problem is the *virtualization of sink and source spaces*. The *virtual space* is constructed by a set of memory modules together with *working disks*. One working disk is associated with each memory module, and behaves itself in the very similar way to the swapping disks used in the virtual memory management in the conventional operating system.

Before we proceed, the *hierarchy of clusters* is first introduced (Figure 8). In the hierarchy, we refer to a cluster by an *atomic cluster*. By cluster size tuning, some number of atomic clusters are grouped to a *processing cluster*, which fits in one processing module. The processing cluster is the unit of data which can be processed in one processing module.

Usually, (sink or source) memory modules are to store the entire set of atomic (hence processing) clusters. However, in case the very large data stream resulted, or enough number of memory modules is not available because of the lack of resources, we consider the larger unit of data whose size is equal to the size of the total capacity, of (sink or source) memory modules allocated to the task. The set of atomic clusters which fits in the set of (sink or source) memory modules in a task is called *staging cluster*. When the result data stream exceeds the total capacity of the allocated memory modules, more than one staging clusters form the entire data stream.

Now, we return to the overflow problem. If the data stream overflows in the sink memory modules, first we consider the stream as the set of staging clusters. In the virtual space, these staging clusters are arranged so that one of them are stored in the memory modules, while others are *destaged* to the working disks associated with each memory module (Figure 9). Consider the data transfer from processing modules to memory modules in an internal task. While receiving the result data streams, memory modules occasionally destage some atomic clusters to their working disk modules if the overflow is expected to occur. When the transfer of data completes, the memory modules are filled with data which form the staging cluster to be executed first in the next operation, whereas the working disks store the remaining data without losing the clustering property. To execute the subsequent task, the sink space is changed to the source space, then begins to output the parallel data substreams from memory modules in it. As the data in memory modules are output, the new staging clusters are successively staged up to the memory modules

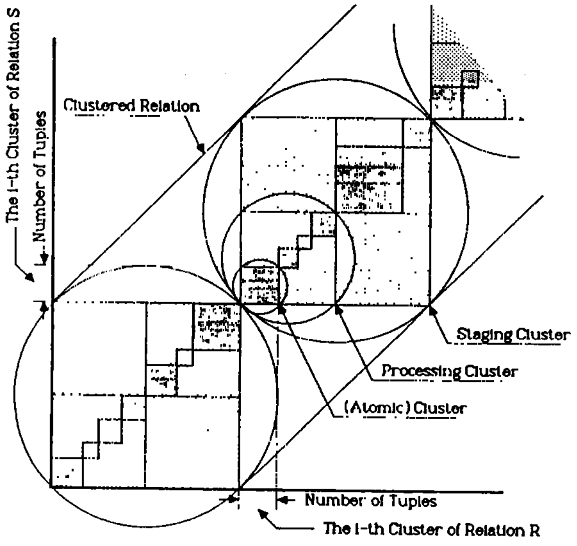


Figure 8. Cluster Hierarchy

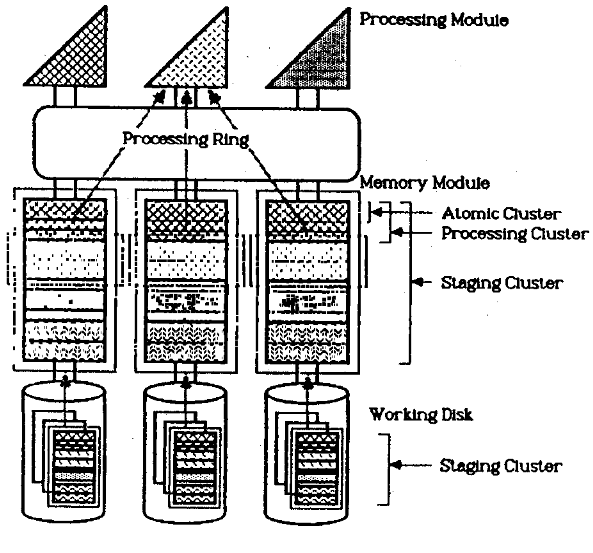


Figure 9. Virtual Space Management

from the working disks. Thus, when some staging cluster is consumed, the next one is formed at the same time in memory modules. This means that the data are continuously supplied from the working disks to the memory modules in the pipelined fashion. Consequently, the memory modules, together with the working disks, generate a clustered very large data stream without serious overhead (that is, without thrashing between disks and memories). Note that the perfect estimation of the access pattern of data is naturally achieved by this cluster-based scheme.

Which atomic cluster should be destaged? Memory modules in the sink space select the *largest atomic cluster at that time*, because such a cluster tends to become much larger, hence might exceed the capacity of the processing module. Such a rather large cluster will be further clustered using finer hashing function in working disk modules.

Note that we still keep the linear time execution of tasks even if the spaces are virtualized. This is quite different from the virtualization scheme in RAP [Ozka77] which reveals the squared order of execution time.

3. System Software Organization

The organization of system software we propose consists of four major components, which are organized in a layered hierarchy. Figure 10 shows an overview of the components we will discuss. At the heart of the system software lie the *primitive (hardware) algorithms* employed in disk or processing modules. They implement filtering and clustering functions in tasks. For example, hardware sorter, together with tuple manipulation unit, implements the filtering function in an internal task. Softwares in the next layer constitute the *intra-task control* that implements the efficient flow control of data streams *within* each task. In other words, it enables streams to act autonomously. Just beyond the intra-task control comes the *inter-task control layer* that controls the initiation and deletion of cooperating tasks, such as the resource management, scheduling of the execution of tasks, task firing, and deadlock resolution. The top level layer is the *transaction management*, which produces the executable task trees and makes transactions atomic and recoverable activities.

The four layers of the system software are implemented in GRACE in the fully distributed manner (Figure 11). The primitive algorithms layer is implemented by its hardware modules. The intra-task control layer is placed in the ring bus interface units of modules, thus implemented in the form of *protocol* between them. The inter-task control layer is on two control modules in cooperation with ring bus interface units of modules. The transaction management layer is implemented mainly on the control module on the staging ring.

The primitive algorithms layer was described in the previous section as the query processing algorithms. We present the intra- and inter-task control layers in the following sections. The transaction management layer, along with the concurrency control, is briefly described in section 7.

4. Intra-task Control Layer

Intra-task control layer is responsible for the data stream control in a task. Its objective is to achieve the continuous flow of data in a task. In other words, it provides

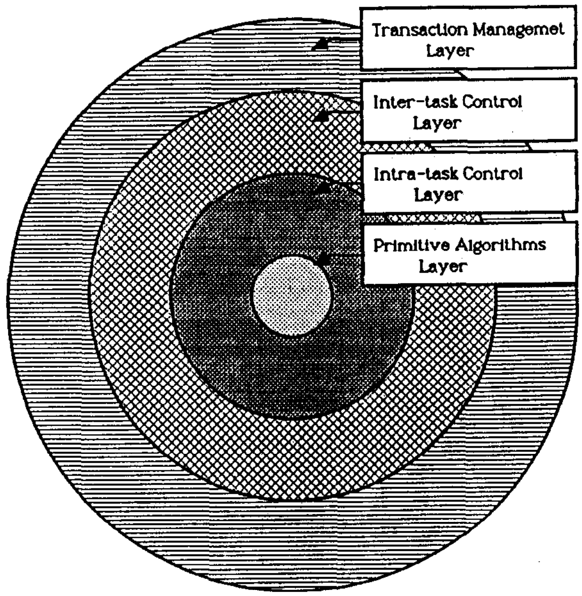


Figure 10. Organization of System Software

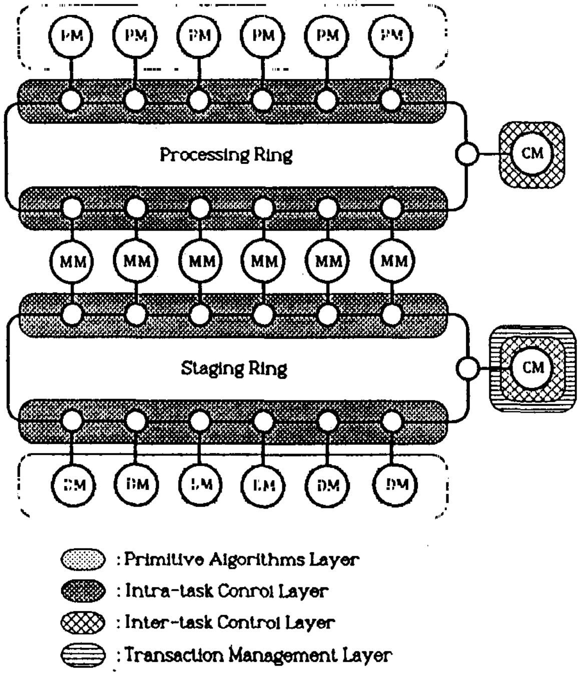

PM
PM
PM
PM
PM
PM
Processing Ring
CM
MM
MM
MM
MM
MM
MM
Staging Ring
CM
DM
DM
DM
DM
DM
DM
: Primitive Algorithms Layer
: Intra-task Conrol Layer
: Inter-task Control Layer
: Transaction Management Layer

Figure 11. Layer Placement

destination modules to each tuple in a stream without causing any overhead. Because of our design policy that a task should support itself, this layer is implemented in the form of protocol, as opposed to the centralized control. Most of efforts were concentrated on the design of this layer because of its outstanding importance in the data stream oriented processing.

Before we proceed, it should be mentioned how parallel task executions are isolated each other. Task executions are fully isolated in GRACE in the following way. All modules keep the id of the task they are belonging to in their ring bus interface unit. Modules respond to channels only if channels have the same task id as modules. Within a task, modules are identified by their relative positions[2], and these positions are used to establish the connection between senders and receivers of data.

4.1. Subtask: Intra-task Structure

Here we examine the flow patterns of streams within a task, and introduce the lower structure of a task called *subtask*.

A data stream in a task is dichotomized according to its flow patterns. In any kind of tasks, tuples are transferred in either of two ways: tuples in each atomic cluster are *distributed* uniformly over destination modules, or, tuples in each cluster thus distributed are *collected* again into some destination module. These are later referred to as *cluster distribution* and *cluster collection* respectively.

According to this criterion, a task itself is decomposed into *subtasks*: a *distribution subtask*, and a *collection subtask*. Then, each task consists of at most two subtasks: a read task consists of one distribution subtask (data transfer from disk modules to sink memory modules), while an internal task is composed of one collection subtask (data transfer from source memory modules to processing modules) and one distribution subtask (data transfer from processing modules to sink memory modules).

4.2. Protocols

In the collection subtask, the module which is to send a tuple can easily determine the destination module by itself as described in section 2.2 (see Figure 7), hence the data streams of this type are rather easy to control; it is sufficient for sender modules just to attach the destination address (the relative position) to the tuple, and then put it on the ring bus. Destination modules in turn always keep an eye on such channels. If matching is found between id's of the module and of some channel, the data on the channel are taken in. The protocol is straightforward. Channels used by this protocol are called *collection channels*.

On the other hand, a distribution subtask is rather difficult to control. In this subtask, tuples in each cluster should be distributed over receiver modules uniformly. To keep the cluster distribution uniform, the destination should be determined so that adding the tuple to it does not cause any skew in the flat distribution achieved so far. Note that the mapping between senders and receivers should be one-to-one because receivers can receive at most one tuple at one time.

Here we present the overview of the protocol for the distribution subtask. As the readers will see, this protocol makes it possible for tuples to be transferred *without any overhead* concerning the protocol execution.

The protocol consists of three phases. Consider the data transfer in a distribution subtask in an internal task as a representative of this subtask. For simplicity, suppose n processing modules are to send n tuples to n memory modules by n channels. Channels used for this type of data transfer are called *distribution channels*. Their formal consists of *header part* and *data segment part*. The header part holds several kinds of control information. The data segment part carries the fixed size of data, or *segment* of tuples.

In the *initial phase*, each of processing modules reserves any one of channels by writing its id on the channel, and at the same time, puts on it the hash value (cluster id) of the tuple to be transferred. Channels thus initialized are then directed to memory modules. In the initial phase, memory modules in turn set the maximum and minimum values among sizes of fragments of the cluster the hash id the channel specifies. That is, if the channel specifies the i-th cluster, $\max_i$ and $\min_i$ among B_{ij} $(j = 1,\dots n)$ are set on the channel by memory modules, where B_{ij} is the fragment size of the i-th cluster stored in the j-th memory module. These two values are used to establish the mapping in the following phase.

Then, in the *link phase*, we establish the one-to-one connection between processing modules and memory modules using such statistics. In the link phase, each processing module puts the first segment of tuple on the channel it reserved in the initial phase. Then, these channels are sent to memory modules. Memory modules in turn examine channels one by one, and select the "best" tuple to take in it. The "best" tuple is determined by the following evaluation function R of B_{ij}, $\max_i$, and $\min_i$ as above:

$$R(B_{ij},\max_i,\min_i) =$$

$$\begin{cases} 1.0 & \text{if } B_{ij}=\min_i \text{ and } B_{ij}=\max_i \\ \infty+\max_i - B_{ij} & \text{if } B_{ij}=\min_i \text{ and } B_{ij}\neq\max_i \\ \min_i - \max_i & \text{if } B_{ij}\neq\min_i \text{ and } B_{ij}=\max_i \\ \dfrac{\max_i - B_{ij}}{B_{ij}-\min_j} & \text{if } B_{ij}\neq\min_i \text{ and } B_{ij}\neq\max_i \end{cases}$$

The value of $R(B_{ij},\max_i,\min_i)$ becomes more than one, if the j-th memory module has less tuples of the i-th cluster than the average, and less than one, otherwise. The value of R indicates the relative suitability on acceptance of a tuple: the greater R value the tuple has, the better the module to take it in.

Memory modules use this function in the link phase to decide the tuple to receive in the following way. Memory

[2] Each kind of modules in one task is successively numbered beginning with one, from downstream to upstream, on the ring. This number is referred to as the *relative position* of the module in the task, and used to identify the module in a task (module id). It should be relative because the physically (or absolutely) adjacent module may be allocated to other task.

module is equipped with a buffer store of one channel size. The buffer is initialized empty at the end of initial phase. In the link phase, every time memory module encounters a channel allocated to the distribution subtask, the function R is evaluated, and the result value is compared with that of the tuple temporarily stored in the buffer: Suppose the j-th memory module encounters the channel carrying the tuple belonging to the i-th cluster. Then, the memory module fetches B_{ij} from its own cluster fragment size table, max_i and min_i from the channel, and then evaluates $R(B_{ij}, max_i, min_j)$. If the value is greater than that of the tuple in the buffer, the contents are exchanged. When the module examines all of channels, the tuple whose atomic cluster gave the greatest result value is left in the buffer. Memory module establishes the connection with the processing module which sent this tuple by remembering its id.

Note that not all channels are examined by a memory module: the first memory module can examine all of channels in the subtask, while the second module examines all of channels except one which is already taken in by the first. In general, the i-th memory module selects the tuple out of the candidates except those which are taken in by its preceding $i-1$ modules. In particular, the last memory module has no choices. In this scheme, therefore, the more upstream the module is located, the more preferential treatment is given. The evaluation function R was carefully designed not to make the modules downstream disadvantageous.

The example of the link phase is shown in Figure 12. Note two id's used to designate the processing module in the figure. One is used to specify the "owner" of the channel, while the other indicates the "sender" of data in the channel. These two id's should be used because of the exchange operation in the link phase.

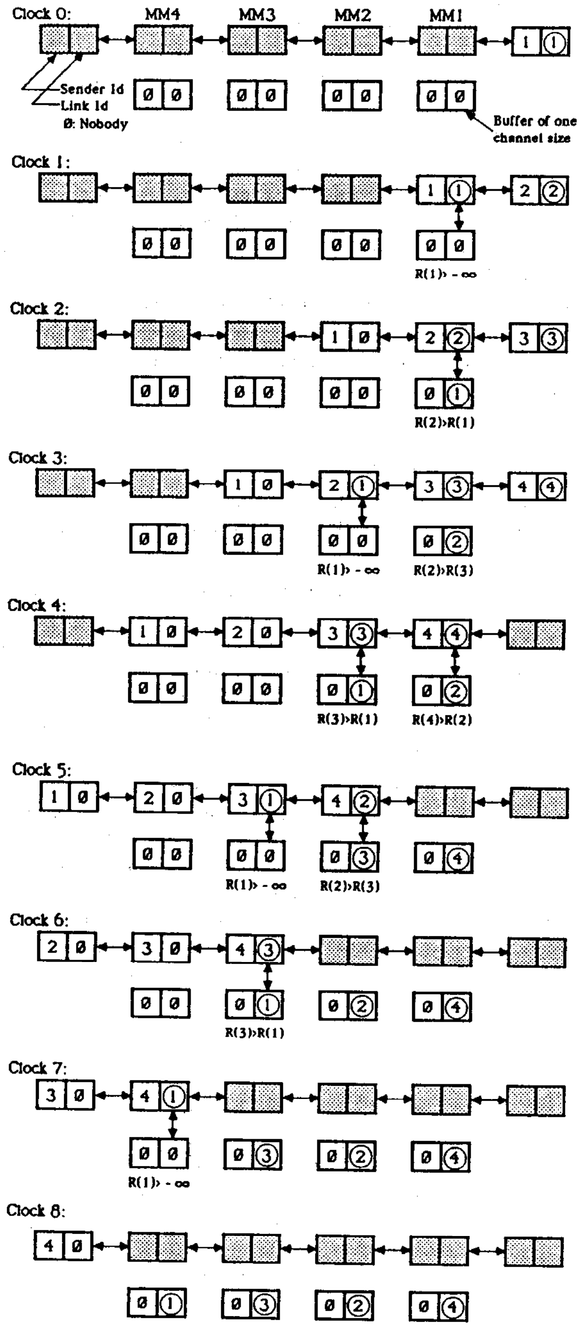
The link phase is followed by the *transmission phase* in which remaining segments are transferred through the connection thus established. Processing module puts the remaining fragment on the channel it reserved, while memory module takes in the data which are sent by the processing module it is connected to. This phase is repeated necessary times to transfer a whole tuple.

In the actual implementation, the initial phase is overlapped with the transmission phase in which the last data segment of the previous tuple is transferred. The net effect is that the initial phase is completely hidden behind the transmission phase, hence no explicit overhead exists.

The protocol has been generalized to handle the general case in which the dynamically variable number of tuples are sent to the receiver memory modules which may occasionally refuse to receive tuples when they become too busy with other jobs like working disk management. The details on it can be found in [Fush86].

5. Inter-task Control Layer

In our design, it should be kept in mind that the control module is consulted by other kind of modules much less frequently than usual. This is because the operand granularity for enabling modules is much coarser and larger than other designs, and also because cooperating modules work quite autonomously in the form of task: once tasks are activated, they can run without any centralized control. The indication is that one control module on each ring would be able to manage the machine without causing serious overhead.



R(n) : R(B$_{ij}$, max$_i$, min$_j$) computed at the j-th memory module for the i-th cluster sent from the n-th processing module (i and j are not explicitly given in the figure.)

Link Id : The id of processing module which owns a channel
Sender Id : The id of processing module which sent the data on the channel

Figure 12 Link Phase in Distribution Protocol

The main part of the inter-task control layer is implemented on these two control modules. The control module on the processing ring is responsible for the control of internal tasks, while that on the staging ring is in charge of managing read tasks. These two control modules perform same management routines except the conflict/deadlock resolution on the control module on the staging ring [Fush86].

Roughly speaking, the inter-task control manages executions of tasks by repeating the following procedures. It first identifies *firable* tasks. By firable task, we mean the task whose child tasks in the task tree finished, hence its source space is ready to output data streams. Next, it selects one of firable tasks, and allocates the sufficient resources to it to enable its execution. Then, it exactly fires the task. The control can set about the next firable task without waiting for the completion of this task. Eventually, the inter-task control is noticed the completion of the task execution. Upon completion, it deallocates the resources, and removes the task from the system.

There are thus four main components in the inter-task control layer; *resource management*, *channel allocation management*, *task monitor*, and *channel deallocation management*. They are invoked when some specific channel is encountered. Throughout this section, we consider the execution of an internal task for the illustrative purposes.

5.1. Resource Management

In the design, resources are managed in a rather novel manner: they are managed *without* using any kind of resource management table. The scheme is based on the following idea which utilizes the channel regulation property of ring bus: resource manager puts (sub)task id on the *(sub)task id field* and the numbers of necessary processing and memory modules on the *module count fields* of the special channel, requesting participation of modules in a task. The idle module in turn responds to such a channel by decrementing the corresponding count field of the channel to notify it intends to join the task. The channel is eventually returned to the resource manager, and its fields are examined. If all of its module count fields are decremented to zero, desired number of modules are reserved. Otherwise, the system currently lacks modules. This protocol will be later referred to by the *count-down protocol*, and the channel used for resource management purposes in that way is called *allocation channel*. Only one allocation channel is available in the system. The resource management routine which is responsible for allocating modules to a task is referred to by the *resource manager*.

To allocate resources to a task, resource manager requests available modules to join the task. More precisely, it uses three module count fields on the allocation channel to specify the number of memory modules in source space, processing modules, and memory modules in sink space. The allocation channel also carries the collection and distribution subtask id's. Note that the common distribution subtask id is used among the sibling tasks. All what is required for the execution of tree-formed tasks is to let sink memory modules know this common id. The protocol for distribution subtask automatically merges result data streams from multiple tasks.

Basically, a task is allowed to be fired only when these three module count fields are decremented to zero after the count-down protocol is executed. We can proceed the task activation, however, even if expected size of sink space is not reserved (the module count field for sink memory modules is not decremented to zero), because of the virtualization mechanism described in section 2.4.

When the allocation channel is returned, resource manager informs modules temporarily assigned to the task of success/failure of resource allocation again by the count-down protocol. Modules then initializes themselves for the task execution, or return to the previous state.

5.2. Firing Tasks

The *channel allocation management* is responsible for allocating necessary number of channels to the tasks which have been already allocated modules by the resource manager. A task is actually fired just when the channel allocation completes.

Whenever an empty channel is shifted in the control module, the channel allocation management is called. If there is a task waiting for the channel allocation, the empty channel is formated either as the collection channel, or distribution channel for it. When all of channels are allocated for the task, now the task is actually fired. No directions to enable modules are required. The modules are automatically activated under the intra-task control protocol just when they find these channels in the task in which they are involved.

5.3. Execution Monitoring

Once task is fired, its execution status is monitored by the *task control channel*.

The task control channel carries enough information on the execution status of active tasks, including the current number of source memory modules, processing modules, and sink memory modules. The count-down protocol is again used in the task control channel: modules which finished sending/receiving the data stream decrement the corresponding module count field. These values are examined by the *task monitor* in the inter-task control every time the task control channel is directed in the control module. When all of these fields are decremented to zero, the task monitor directs source memory modules and processing modules to leave the task. As for the sink memory modules, it examines whether the task is the last child among siblings. If other sibling tasks are still active, or not yet activated, only the source memory modules and processing modules are freed, while sink memory modules are kept allocated to absorb the data streams from its siblings which are not yet complete. Otherwise, the task monitor requests sink memory modules to leave the task.

5.4. Task Termination

To wind up the task, channel deallocation is finally performed. The following routine, *channel deallocation management*, is responsible for it. This routine is called every time the control module finds the collection channel or distribution channel. According to the type of the channel, it checks if the corresponding (sub)task terminates. If so, it empties the channel. When all of channels in a task are deallocated, a task is exactly finished.

6. Software Simulator

The system software described in the previous sections has been implemented on the software simulator of GRACE [Fush86], and several performance evaluations are being conducted. We are primarily concerned with the performance of intra- and inter-task control layers: how well the control module manages the execution of tasks, and how continuously data streams flow through the machine. Note that the former is is concerned with overhead before a task is fired or after it terminates, while the latter is on the performance during the execution of the task.

Preliminary performance evaluation showed that our design enabled the machine to work with little control overhead. The implementation of the routines in the inter-task layer were very efficient. In the implementation, tasks waiting for several services such as resource allocation form a queue. Thus, all what is required for the layer is to examine a few number of fields on the channel, to fetch information of tasks from the queue, or to modify some fields in the control table. These actions are activated just when the specific channel is shifted in the control module, and can be finished before the channel leaves the control module.

It was also shown that data stream flowed through the machine quite continuously, and that almost perfect data stream oriented processing was achieved. The continuous data streaming is achieved by the complete pipelining in the collection subtasks, which is in turn accomplished by the uniform distribution of tuples in the preceding distribution subtask, along with the cluster size tuning. The protocol for the distribution subtask described in section 4 can achieve the almost perfect uniform cluster distribution for any set of tuples and hash functions. The difference between the fragments of clusters observed so far was negligibly small (only a few number of tuples).

7. Conclusion

In this paper, the system software design of a parallel relational database machine GRACE was described. Its preliminary implementation on the software simulator of GRACE was also reported. The preliminary implementation showed that our design enabled the machine to work with little control overhead, and that the almost perfect data stream oriented processing was achieved. The novel virtual space management algorithm was also proposed.

Here, we present the brief description of the concurrency control and transaction management in GRACE. The concurrency control is also data stream oriented; it uses the precision lock [Jord81] in which the read access is performed by the predicate, while the write access is carried out by the units of tuple. Disk module in GRACE has another filter processor, and uses it to sieve out the conflicting updated tuples. The predicates issued by the transactions which do not yet commit are set in this filter processor. Thus, the conflict check can be carried out in the same way as selections: a tuple in an updated data stream causes conflict if it satisfies some predicate in this conflict check filter. When conflict is detected, it is reported to the control module on the staging ring, and resolved there.

For recovery purposes, this filter processor also collects the before and after values of updated tuples. These logs are stored in the local log disk in the disk module. The control module on the staging ring, on the other hand, stores the begin and commit transaction log records in its own commit log disk. To commit the transaction, the control module requests disk modules to flush the log data on the transaction. When all of logs are successfully flushed, the commit log is recorded in the commit log disk. To recover the system after a crash, the *recovery manager* on the control module on the staging ring can perform winner/loser analysis only by examining the commit log disk. Subsequent undo's and redo's can be executed in parallel by disk modules. In addition, our system software allows the parallel execution of transaction steps *within* a transaction. This requires the extended theory of the serializability. The details on the concurrency control and transaction management, along with this extended serializability, can be found in [Fush86].

Besides conducting the detailed performance evaluation of the system software, we are currently developing the "core" of GRACE, the functional disk system [Kits86], and hash-based software relational database system. The functional disk system is a movable head disk with "functionality". Its current configuration consists of one SMD disk drive, the disk controller with enhanced functionality, and two 68000 micro processors. The disk controller implements the on-the-fly selection and dynamic clustering based on hashing. As mentioned before, the performance improvement observed so far is drastic.

The hash-based data stream oriented *software* relational database system is also being implemented on the conventional mini-computer. The idea is to implement the "miniature" of GRACE by means of cooperating concurrent processes. The virtual space management algorithm described in section 2.4 makes it possible for this system to handle large databases with limited memory capacity. The details will be reported elsewhere, along with the overall performance of GRACE.

References

[Bora80] Boral, H. and DeWitt, D. J., "*Design Considerations for Data-Flow Database Machines*", Proc. of the ACM-SIGMOD International Conference on Management of Data, pp.94-104 (1980)

[Bora82] Boral, H., et al, "*Implementation of the Database Machine DIRECT*", IEEE Trans. on Software Eng., SE-8(6), pp.553-543 (1982)

[Cham76] Chamberlin, D. D., et al, "*SEQUEL 2: A Unified Approach to Data Definition, Manipulation, and Control*", IBM Journal of Research and Development, 20(6), pp560-575 (1976)

[DeWi78] DeWitt, D. J., "*DIRECT - A Multiprocessor Organization for Supporting Relational Database* Management Systems" Proc. of 5th Annual Symposium on Computer Architecture, pp.182-189 (1978)

[DeWi86] DeWitt, D. J., et.al, "*GAMMA: A High Performance Dataflow Database Machine*" Technical Report, #635, Univ. of Wisconsin-Madison (1986)

[Jord81] Jordan, J. R., Banerjee, J., and Batman, R. B., "*Precision Locks*", Proc. of the ACM-SIGMOD International Conference on Management of Data, pp.143-147 (1981)

[Kim84] Kim, W., Gajsk, D., and Kuck, D. J., "*A Parallel Pipelined Relational Query Processor*", ACM Trans. on Database Systems, 9(2), pp.214-242 (1984)

[Kits82] Kitsuregawa, M., Tanaka, H., and Moto-oka, T., "*Relational Algebra Machine GRACE*" in Lecture Notes in Computer Science, Springer-Verlag, pp.191-214 (1982)

[Kits83] Kitsuregawa, M., Tanaka, H., and Moto-oka, T., "*Application of Hash to Database Machine and Its Architecture*", New Generation Computing, 1(1), pp.63-74 (1983)

[Kits84] Kitsuregawa, M., Tanaka, H., and Moto-oka, T., "*Architecture and Performance of Parallel Relational Database Machine GRACE*", Proc. of International Conference on Parallel Processing (1984)

[Kits85] Kitsuregawa, M., Fushimi, S., Tanaka, H., and Moto-oka, T., "*Memory Management Algorithms in Pipeline Merge Sorter*", Proc. of International Workshop on Database Machines (1984)

[Kits86] Kitsuregawa, M., and Takagi, M., "*Performance Evaluation of Functional Disk System*", to appear in Proc. of ICCD 86 (1986)

[Fush85] Fushimi, S., Kitsuregawa, M., Tanaka, H., and Moto-oka, T., "*Algorithm and Performance Evaluation of Adaptive Multidimensional Clustering Technique*", Proc. of ACM-SIGMOD International Conference on Management of Data (1985)

[Fush86] Fushimi, S., "*System Software of A Relational Database Machine Based on Data Stream Model*", Ph.D Thesis, University of Tokyo (1986)

[Ozka75] Ozkarahan, E. A., Schuster, S. A., and Smith, K. C., "*RAP - An Associative Processor for Database Management*", Proc. of AFIPS NCC, 44, pp.379-387 (1975)

[Ozka77] Ozkarahan, E. A. and Sevcik, K. C., "*Analysis of Architectural Features for Enhancing The Performance of A* Database Machine", ACM Trans. on Database Systems, 2(4), pp.297-316 (1977)