PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1984,
edited by ICOT.

MPDC : MASSIVE PARALLEL ARCHITECTURE FOR VERY LARGE DATABASES

Yuzuru Tanaka

Department of Electrical Engineering
Hokkaido University
Sapporo, 060 JAPAN

ABSTRACT

Massive Parallel Database Computer (MPDC) is a relational database machine architecture that integrates microparallel VLSI architectures for basic relational operations and macroparallel data flow control architecture for coordinated concurrent execution of subtasks. MPDC consists of Data Subsystem and Control Subsystem. Data Subsystem is in charge of segment accesses and segment processing, while Control Subsystem is responsible to Data Subsystem for decomposing query transactions into concurrently executable segment processing commands. Data Subsystem consists of a pool of processors for segment processing, a set of disk subsystems, and a multiport page buffer that is shared by these modules. Segment processors embody microparallelism of segment processing, while Shared Page Buffer resolves resource conflict problem in parallel processing and allows massively parallel processing based on macroparallelism among segment processing tasks. Control Subsystem uses a unified control algorithm that does not only manage adaptive segmentation of relational files but also efficiently and correctly control highly-reliable interleaved execution of transactions. Activation of segment access commands and segment processing commands are controlled by a data flow controller, which automatically controls disk subsystems to taransfer segments to Shared Page Buffer prior to the processing of them.

1 INTRODUCTION

'Most organisms on earth depend on their genetic information to a much greater extent than they do on their extragenetic information. For human beings, it is the other way around. We have through our brains, a much richer opportunity to blaze new behavioral and cultural pathways on short time scales. In addition, human beings have, in the most recent few tenths of a percent of our existence, invented not only extragenetic but also extrasomatic knowledge information stored outside our bodies, of which writing is the most notable example.' This is quoted from 'The Dragons of Eden' by Carl Sagan (Sagan 1977). Extrasomatic knowledge accompanied by extrasomatic processing and extrasomatic reasoning will give us far richer opportunity to blaze newer behavioral and cultural pathways on much shorter time scales. This effect will be accelerated by appearances of high performance database machines and knowledge base machines that can cope with very large databases or knowledge bases.

While current super computer systems aim at high speed numerical computations, future super systems coping with extrasomatic information and extrasomatic reasoning will require vast amount of computing power to produce appropriate information from a huge information reservoir by repetitive retrieval and reasoning. The arrival of such future super systems with vast amount of computing power for database and knowledge base processing requires innovations in the following technologies:

(1) VLSI architectures for high speed processing of primitive functions that are fundamental in database or knowledge base processing.

(2) Hierarchical shared memory organization that allows concurrent accesses from massively parallel processors.

(3) File clustering schemes that increase access locality and decrease file access frequency.

(4) Control mechanisms for cooperative coordination of massively parallel processes.

While high speed on-core processing is most important in current super computers, in future super systems, memory hierarchy and file clustering will become equally or more important than that. The use of moving head disk units is inevitable to provide a sufficiently large storage space, while basic processing in databases or knowledge bases generally requires references to vast amount of data. Every basic processing is likely to access secondary memories repetitively. This

situation should be avoided. Otherwise, disk accesses will introduce serious delay to almost every basic processing.

Previous studies on database machines, however, aimed at the high-speed brute force processing of a full search. Some of them proposed direct search of rotating disk tracks (Coulouris et al. 1972, Babb 1979, Ozkarahan et al. 1975, Chang 1978, Uemura et al. 1980, Schuster et al. 1979, Oflazer et al. 1980). Some others proposed buffer memories and a network that allows arbitrary connections between multiple processors and multiple memory banks to eliminate interprocessor transfer of vast amounts of data (Dewitt 1979). Some others studied VLSI modules for basic database operations (Kung and Lehman 1980, Tanaka et al. 1980). However, few researchers proposed file clustering schemes and hierarchical memory organizations based on them (Banerjee et al. 1978, Tanaka 1983a). No researchers have ever proposed coordinated concurrency control of multiple segment processing tasks obtained by decomposing multiple transactions.

This paper will propose a **massive parallel database computer** architecture **MPDC** with innovative solutions to the above four technological difficulties. The first three were independently solved in my previous papers (Tanaka et al. 1980, Tanaka 1984a, 1984b, 1983b). Two VLSI architectures **Search Engine** and **Sort Engine** proposed in 1980 gave a solution to high speed processing of basic functions (Tanaka et al. 1980). Recently in 1984, they are modified to allow **bit-slicing** (Tanaka 1984a). Hierarchical shared memory organization was solved in 1984 by a **multiport page-memory** architecture that allows 10^3~10^4 concurrently accessible ports without causing any conflict nor any suspension (Tanaka 1984b). File clustering was solved by **colored binary trie schemes** proposed in 1983 (Tanaka 1983b).

This paper will give a brief survey on these technological breakthroughs and will propose an overall massive parallel database computer architecture and its **coordinated control** structures that effectively integrate the independent fundamental technologies. Section 2 will briefly explain architectural philosophy of MPDC and an outline of MPDC architecture. MPDC consists of two subsystems, i.e., **Data Subsystem** and **Control Subsystem**. Data Subsystem is in charge of segment accesses and segment processing, while Control Subsystem is responsible to Data Subsystem for decomposing query transactions into concurrently executable segment processing commands. Two types of parallelism will be distinguished. Parallelism in each basic segment processing will be referred to by **microparallelism**, while parallelism that is foud in concurrent processing of segment accesses and segment processing in interleaved execution of multiple transactions will be called **macroparallelism**. Section 3 will describe Data Subsystem architecture. It consists of a pool of processors for segment processing, a set of disk subsystems, and a multiport page buffer that is shared by these modules. Segment processors use microparallelism in their pipeline architecture, while Shared Page Buffer has resolved resource conflict problem in parallel processing and allows massively parallel processing based on macroparallelism. Section 4 will describe Control Subsystem architecture together with two important algorithms that effectively and correctly increase macroparallelism in multiple transaction processing, i.e., file clustering and concurrency control algorithms. This section will emphasize the importance of a unified control algorithm that does not only manage adaptive segmentation of relational files but also efficiently and correctly control highly-reliable interleaved execution of transactions. Activation of segment access commands and segment processing commands are controlled by a data flow controller, which automatically controls disk subsystems to taransfer segments to Shared Page Buffer prior to the processing of them.

2 AN OUTLINE OF MPDC ARCHITECTURE

2.1 MPDC Design Philosophy
Microparallelism & Macroparallelism

High-volume processing of databases requires frequent references to a very large storage space, which makes it inevitable to frequently access mechanically-accessed secondary memory devices like moving head disk units. Database machine research efforts are now confronted with 'Disk Paradox' as pointed out by H. Boral and D. J. Dewitt (Boral and Dewitt 1983). Obviously, the number of disk units necessary to store a database is inversely proportional to the capacity of a single disk unit. Hence, the maximum outflow obtainable from a set of disk units is proportional to the transfer rate of a single unit, and inversely proportional to its capacity. On the other hand, the current disk development effort is directed toward increasing disk capacity with little improvement or even deterioration in transfer rate. This lowers the maximum outflow from secondary memories, which bounds machine performance and nullifies speed-up technologies of on-core processing.

Historically, this kind of problems has been repetitively encountered, and resolved through the combination of two technologies. A buffer memory placed between a primary memory and a secondary memory does not only increase access speed but also decreases secondary memory access frequency, while data clustering into segments increases access

locality, and does not only decrease segment references but also enhances the buffer effect by increasing a chance of repetitive references to a small set of segments.

These two techniques necessarily introduce a secondary memory access unit called a segment. Segmentation divides database processing into two processing levels, i.e., segment search and segment processing. For a given transaction, segment search searches file directories to generate a set of segment processing commands with one or two segment locations as operands. Segment processing, on the other hand, executes, for each segment command, a basic database operation on one or two operand segments. It requires to fetch operand segments from disks to a work space if they are not there yet. Decomposiion of a given transaction into segment processing commands must be controlled by a well-defined concurrency control scheme to maintain database integrity.

Speed-up of database processing requires massively parallel processing at each processing level. Parallelism in segment search will be referred to by **macroparallelism**, while parallelism in segment processing will be called **microparallelism**. These two types of parallelism are inherently different. The main concern of microparallel processing is the high-speed processing of each basic operation on one or two segments. Macroparallel processing, on the other hand, aimes at massively parallel processing of concurrently executable segment operations. It must cope with transaction decomposition, concurrency control of interleaved transaction execution, and segment command generation. Since microparallelism concerns parallelism in each basic operation, it has a definite structure that can be apriori described. Macroparallelism, on the other hand, concerns parallelism among tasks each of which represents a segment operation. Therefore, the structure of macroparallelism depends on less definitely describable factors like the status of concurrently executed transactions and the status of each computer resource. Definite structures of microparallelism can be embodied by definite parallel or pipeline algorithms that are suitable for VLSI implementation. Macroparallelism, however, requires much flexibility in parallel processing control. No control architectures other than data flow control may have the required extent of flexibility.

2.2 An Outlined Architecture

MPDC has a configuration as shown in Fig. 2.1. It consists of two subsystems, i.e., Control Subsystem for segment search and Data Subsystem for segment processing.

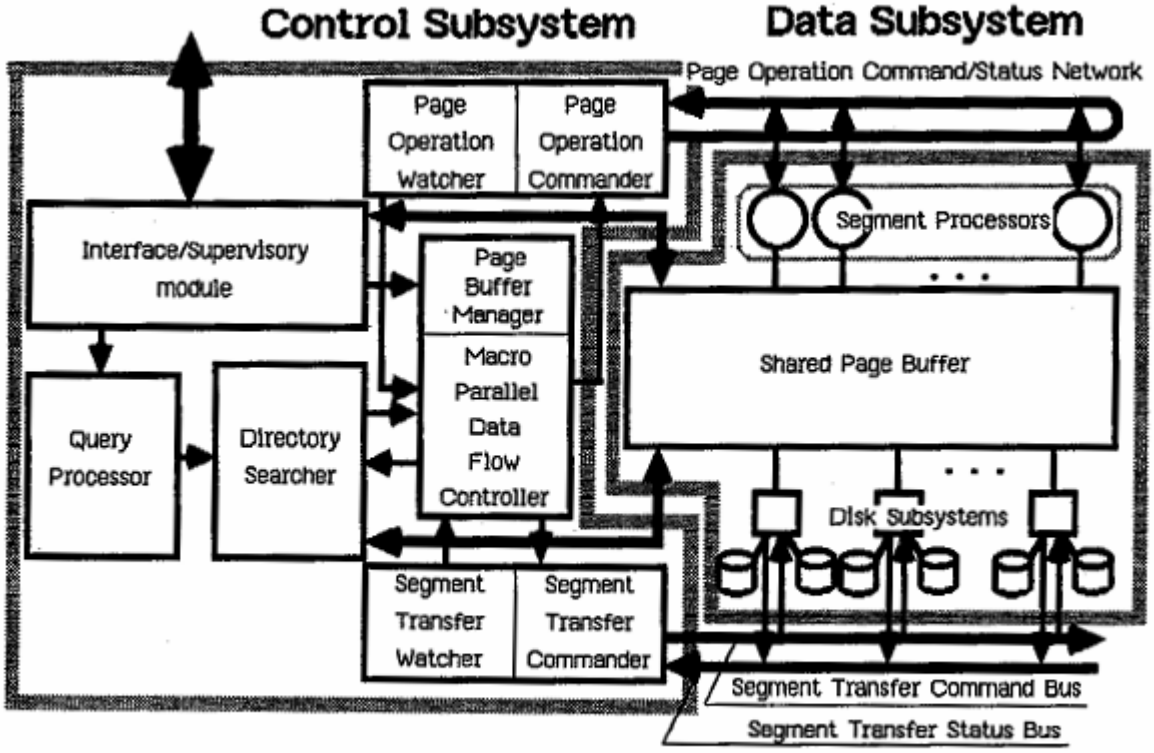


Fig. 2.1. Hardware Configuration of MPDC.

Control Subsystem receives user queries, analyzes and decomposes them into segment processing operations through directory searches, and generates concurrently executable segment commands, while Data Subsystem receives generated segment commands from Control Subsystem, executes segment commands concurrently, and sends back a completion token to Control Subsystem immediately after the completion of each command execution.

Data Subsystem consists of a homogeneous set of segment processors, a set of disk subsystems, and a shared page buffer between these two sets of devices. Each disk subsystem consists of a disk controller and several disk units. Shared Page Buffer is a page-access memory that is shared by the set of processors and the set of disk subsystems. It is divided into equal-sized pages. Each page can store one segment of a file and has a unique page address. Shared Page Buffer allows all devices connected to it to concurrently access arbitrary pages without causing any access conflict nor any access wait.

Segment commands in MPDC are classified into two categories, i.e., segment transfer commands and page operation commands. Examples of segment transfer commands are get and put commands. They have two operands, i.e., a segment address and a page address. Each segment in disk subsystems has a unique segment address, which uniquely determines the disk unit that stores this segment. This unit is called the home disk of this segment, while the disk subsystem with the home disk is called the home disk subsystem. A get command requests Data Subsystem to read out its operand segment from the home disk to its operand page in Shared Page Buffer, while a put command requests Data Subsystem to save its operand page value stored in Shared Page Buffer into the operand segment in its home disk.

A page operation command, on the other hand, requests Data Subsystem to execute a relational database operation on its one or two operand pages, and to write one or more pages of result in Shared Page Buffer location specified by its destination operand. These source operand pages must be already loaded either with some segments by get commands or with intermediate result by another page operation command.

Each segment transfer command is sent by Segment Transfer Commander to its operand segment's home disk subsystem through Segment Transfer Command Bus. Its home disk subsystem, when it has received the command, accesses the home disk and transfers one page of information to or from Shared Page Buffer. When a transfer has finished, the home disk subsystem writes a completion status code in its status port, and sends a one bit signal to Segment Transfer Watcher, which always watches segment transfer completion signals and, when one of them is set, gets the associated completion status code through Segment Transfer Status Bus. This completion status code is used as a token by Control Subsystem to activate next executable commands.

Each page operation command may be executed by any processor in the large pool of segment processors. It is put on the ring network (called Page Operation Command/Status Network) by Page Operation Commander and is circulated among segment processors. The first encountered idle segment processor takes out this command from the ring network to execute it. In execution of a three operand page operation command, for example, the allocated segment processor reads out two pages from Shared Page Buffer one after another, executes the operation, and stores the result into the destination operand page in Shared Page Buffer. If a page is not full, the segment processor is not required to read the full page. Actually, each page in Shared Page Buffer is divided into equal-sized tracks. Segment processors need not read unnecessary tracks, but it must read all words in a necessary track. When the segment processor has finished its execution, it sends out its completion status code into the next empty packet circulating on Page Operation Command/Status Network. Page Operation Watcher always watches circulating packets on the network. Immediately upon receiving a completion status code, it sends this as a token to the data flow control mechanism in Control Subsystem.

Segment processors might be considered as eager day-laborers, while Page Operation Commander might be considered as a day-laborers' boss. Disk subsystems and Segment Transfer Commander might be considered respectively as warehouse workers and as a warehouse workers' boss. Shared Page Buffer is a large table used for shipping and discharging. Control Subsystem corresponds to a planning department of a company. The company receives customers' orders. The planning department decomposes these jobs into a set of subtasks. The warehouse workers' boss orders each warehouse worker to ship or discharge cargos that are under this worker's resposibility, while the day-laborers' boss circulates subtask orders among day-laborers. Every jobless day-laborer is very eager to get a job. He will jump at a job when it is circulated to him. The large table is divided into equal sized sections. Warehouse workers carry cargos to and from specified sections. Each day-laborer has his own private work table. A day-laborer, when he has got a subtask order, goes to specified sections to carry cargos from the large table to his private work

table. Then he does the task and produces a result cargo. Finally he carries this cargo to a specified table section and becomes jobless. The large table provides a lot of workers with a common work space for cooperative concurrent achievement of subtasks.

The increase of concurrently executable segment commands is essential in performance enhancement. This is concerned by Control Subsystem. Control Subsystem consists of five major modules, i.e., Interface/ Supervisory Module, Query Processor, Directory Searcher, Macroparallel Data Flow Controller, and Page Buffer Manager.

Interface/Supervisory Module is a super minicomputer that communicates queries and set of data with external systems. It is directly connected to one port of Shared Page Buffer so that it can directly access arbitrary pages in the buffer. Through this path, it can get result pages or initially store a database into MPDC. It works as a service processor to initially store databases in MPDC or to periodically save a dump copy of databases and transaction logs. Besides, it manages transaction statuses.

Query Processor is a minicomputer that transformes a given transaction into an optimized program that searches segment directories and generates segment processing commands. Each segment processing command has one or two source operands s_1, s_2 and the destination operand s_3. Each operand is either a segment of a relational file or a variable. Each variable may be either a single page variable with one page capacity or a multiple page variable with a size of arbitrary number pages. A segment command

$$s_3 \leftarrow \langle\text{relational operation}\rangle\ s_1\ (,s_2)$$

requests both execution of a relational operation on s_1 (and s_2) and the saving of the result in s_3. Programs transformed from queries by Query Processor have lock statements to control interleaved execution correctly.

Directory Searcher is a super minicomputer that receives transformed programs from Query Processor. It executes the programs to search segment directories and to generate segment processing commands. Generated segment processing commands are sent to Macroparallel Data Flow Controller. Directory Searcher has a sufficiently large primary memory and an external low speed semiconductor memory that is large enough to store all segment directories. This external memory is backed up by the disk subsystems in Data Subsystem through a direct connection to one port of Shared Page Buffer.

Macroparallel Data Flow Controller receives segment processing commands, dynamically constructs data flow programs of segment commands consisting of segment transfers and page operations, sends active segment commands to Data Subsystem through the two commanders, receives completion tokens from Data Subsystem through the two watchers, and transfers activation tokens to next executable commands in data flow programs.

Macroparallel data flow programs in Macroparallel Data Flow Controller require each segment to be assigned to a page variable prior to any operation on it. When a segment value is once assigned to a page variable, further references to this segment value refer to this variable. In dynamic construction of data flow programs, segment processing commands sent from Directory Searcher are modified to satisfy this rule. This translation uses a table called Segment Table, which stores information about the assignment of a variable to each segment that has appeared as a source operand. If a source segment s of a sement processing command sent from Directory Searcher has not been registered in Segment Table, Macroparallel Data Flow Controller generates a page variable v and generates a get command

$$v \leftarrow \text{get } s$$

before this command. The reference to s in the original command is replaced by the reference to v. Further references to s are all replaced by references to v. A segment processing command that has a segment s as a destination operand is an update command. Such a command is divided into two segment commands, i.e., one for the assinment of the operation result to a temporary page variable v, and a put command

$$s \leftarrow \text{put } v.$$

Data flow programs can be easily dynamically constructed by renaming variables in the original sequences of segment processing commands to satisfy the single assignment rule, i.e., each variable should not appear more than once as a destination operand. Data dependencies of operand page variables among segment commands are managed by a table called Page Variable Table. For each page variable, this table has an entry that points to a list of page operation and segment transfer commands that refer to this variable as a source operand. The number of elements of this list is stored in the reference count field of the table entry associated with this variable.

Each command list linked to Page Variable Table is used to move activation tokens to next executable commands when the associated variable has been given a page value. Each segment command, whether it may

be a segment transfer command or a page operation command, becomes active whenever it has got tokens for all of its source operands. Each active segment command is sent to Data Subsystem after a page of Shared Page Buffer is allocated to its destination page variable. The destination page variable and its allocated page are registered in Page Variable Table.

The allocation of a page of Shared Page Buffer to a page variable requires page management of Shared Page Buffer. Page Buffer Manager has a memory map that shows, for each page in Shared Page Buffer, whether it is used or free. When it is asked for allocation of a free page either by Macroparallel Data Flow Controller or by one of the segment processors, it searches the map for a free page, sets the corresponding bit of the map, and returns its address to the requesting module.

When Macroparallel Data Flow Controller receives a completion token from Data Subsystem, it decreases the reference count of every source variable of the completed command. When the reference count of a variable becomes zero, this variable is deleted from Page Variable Table, and the page allocated to this variable is made free through Page Buffer Manager.

Some segment commands may require more than one page to save its result. Such a command uses a multiple-page variable as its destination operand. A reference to a multiple-page variable is preceded by an ◇ mark for distinction. Since such multiple page result may become a source operand of another command, multiple-page variables should be allowed to use not only as destination operands but also as source operands.

Macroparallel Data Flow Controller has a table called Multipage Variable Table. When a multiple-page variable is used as a destination operand, a single page is initially allocated to it, and is registered in Multipage Variable Table together with a pointer pointing to a page address list containing only this page. This command is sent to Data Subsystem and is executed by one of the segment processors. If the execution has spent allocated pages and requires more to save the result, the segment processor dynamically asks Page Buffer Manager for one more page through Page Operation Watcher and Macroparallel Data Flow Controller. An allocated page address is sent to the segment processor through Macroparallel Data Flow Controller and Page Operation Commander.

The segment processor remembers the allocated pages, and, when the execution has finished, it sends back the list of allocated pages to Macroparallel Data Flow Controller. This variable length message is sent as follows. The segment processor first writes this list into a page of Shared Page Buffer and sends a completion status containing the address of the page. The list of allocated page is added to the corresponding list linked to Multipage Variable Table.

If a multiple-page variable appears as a source operand of a segment command, Macroparallel Data Flow Controller decomposes this command into a set of commands without multiple-page variables as source operands. The only exception is a 'condence' page operation command, which will be explained later.

3 DATA SUBSYSTEM

3.1 Microparallel Architecture for Segment Processing

3.1.1 Microparallel Architecture

Segment processors perform high speed processing of relational operations on one or two pages of relational files staged in Shared Page Buffer. Page processing requires sequential data transfer of pages between a segment processor and Shared Page Buffer. Large delay caused by sequential transfer is inevitable. To overcome this problem, the segment processor architecture should make much use of transfer time for page processing by overlapping processing with transfer. The overlapped execution of basic functions in database processing with sequential data transfer was first introduced by us in 1980 (Tanaka et al 1980, Tanaka 1982, 1983a). Such a mode of execution was called data stream processing.

The relational model of databases provides a set of database operations as listed below:

set operations :	union, intersection, set difference.
relational operations :	projection, selection, restriction, join, division.
aggregate operations :	count, sum, average, maximum, minimum.
others :	sort.

Suppose that no relations are sorted apriori with respect to some attribute, nor provided with an auxiliary files such as inverted files or link files. Suppose also that the size of each relation is proportional to a single parameter n. Then the time complexity of each operation above is either O(n) or O(n*logn). They are classified as follows:

O(n) : selection, restriction, count, sum, average, maximum, minimum.

O(n*logn) : union, intersection, set difference, projection, join, division, sort.

Operations in the first class can be executed by a single full scan of tuples. The speed up of these operations requires apriori processing of a file such as provision of auxiliary files or segmentation of files. Apriori processing of files also speeds up the processing of the second class operations. However, these apriori processing does not solve the inherent problem: How can we efficiently perform such apriori processing, and the processing of each segment? It is well known that the operations in the second class can be performed in O(n*logn) time when they are executed by algorithms based on sorting. Therefore, sort may be considered as a key function for the speed up of database processing. Besides, we have selected a batch-search operation as an additional key function. Batch search means the batched processing of multiple search processes that search a common table for different search keys. The table is assumed to be apriori sorted. If one of the operand relations is apriori sorted with respect to an appropriate attribute, any binary operation in the O(n*logn) group can be executed more efficiently by a batch search algorithm than by any algorithm based on sorting.

Speed-up of sort and batch search is fundamental in the segment processor design. Some parallel processing algorithms based on data stream processing are required. It is desirable that these algorithms are suited for VLSI implementation. Microparallelism in these basic functions allows satisfactory designs. In our project, VLSI modules embodying high speed data stream processing algorithms of basic functions are called engines. Parallel or pipeline architectures of these engines are referred to by microparallel architectures.

For batch search and especially for sort, there are a lot of VLSI algorithms including those proposed by us. VLSI algorithms suitable for search and sort engines used in each segment processor should satisfy the following requiremwents:

(1) feasible **hardware complexity**,
(2) allowable **pin complexity**,
(3) **large tractable volume** of data,
(4) **data stream processing**, and hence, O(n) processing time,
(5) Wordlength extensibility, i.e., **bit-sliced architectures**,
(6) **tractable data volume extensibility**,
(7) no access to external submodules except I/O, i.e., speed-up by **closed on-chip processing** like on-chip memory accesses.

VLSI search algorithms are classified by three parameters h, d, and t^*, where h denotes the number of comparators, d is a maximum duration time to obtain a search result for a single search key, and t^* is a throughput time of batch search processes, i.e., the time obtained by dividing the total processing time by the number of search keys in sufficiently large batch search. A VLSI search algorithm with h, d, t^* respectively equal to h(n), d(n), $t^*(n)$ for a search table of size n is classified as a (O(h(n)), O(d(n)), O(t^*(n))) type. Search of a table T can be performed by parallel search of smaller tables obtained by equally dividing T. Parallel search using B search modules of a type ($f_1(n)$, $f_2(n)$, $f_3(n)$) forms a ($B*f_1(n/B)$, $f_2(n/B)$, $f_3(n/B)$) type search module. This bank parallelism is applicable to any type of search modules, and hence, it is not considered below. Table 3.1 shows various VLSI search algorithms together with first two software algorithms for comparison. It shows three parameter values, whether a search table is required to be apriori

Table 3.1. Search hardwares

h: number of comparators.
d: maximum processing time for a single search.
t*: throughput time of a batch search, i.e.,

$$\frac{\text{processing time of a batch search}}{\text{number of search keys}}$$

necessity of apriori table sorting
table update capability
bit-slicing

h	d	t*				example
1	n	n		v	v	sequential search
1	$\log n$	$\log n$	v			binary search
h	n	n/h		v	v	multiple-key sequential search
h	$n+h$	$(n/h)+1$		v	v	multiple-key pipeline sequential search
$\log n$	$\log n$	1	v		v*	pipeline batched binary search (Search Engine)
n	n	1		v		pipeline batched sequential search
n	1	1		v		associative memory

v* : Interval Search Engine allows bit-slicing.

sorted, capabilities of table update, and wordlength extensibility.

In database processing, throughput is more important than duration of a single key search. Algorithms with t^* equal to 1 are desirable. They are capable of data stream processing. Among them, necessary comparators should be minimized to minimize hardware complexity, and to maximize tractable data volume of a single chip. Pipelined batched binary search and its extension, pipelined batched interval binary search, have these desirable features. They were proposed respectively as Search Engine (Tanaka et al. 1980) and Bit-Sliced Interval Search Engine (Tanaka 1984a). The latter is the only VLSI algorithm that allows bit-slicing with no external submodule. Since bit-slicing requires at least twice as many connection pins as the number of comparators, it can not be applied to those algorithms with h proportional to n. Two Search Engines can be considered as a pipe that is initially loaded with a serach table and converts each search key of an input stream to its search result during the flow of the stream through this pipe.

VLSI sort algorithms can be classified by different three parameters h, t, D, where h denotes the number of comparators, t is the maximum sorting time, and D is the maximum delay of the first output preparation after the last input. A type (O(h(n)), O(t(n)), O(D(n))) is defined similar to search, where n is the number of values to be sorted. Table 3.2 shows various hardware sort algorithms, including first three software algorithms for comparison. Since sequential input and output are inevitable in high-volume processing microparallel architectures, the total sorting time should be evaluated as the sum of sequential I/O time and the delay D. This desirability measure differs from sorting time t.

Algorithms with fewer comparators and shorter delay are more desirable. Pipeline Heap Sort proposed as Sort Engine has the minimum delay and minimum comparators (Tanaka et al. 1980). The second best in database processing may be Pipeline Two-Way-Merge Sort (Todd 1978). Its O(logn) delay is allowable. This algorithm was extended to allow bit-slicing (Tanaka 1984a). Because of the same reason described above, bit-slicing of those algorithms with h proportional to n, or more than that, is impractical. Besides, pipeline Heap Sort has inherent difficulties in bit-slicing.

These observations show the superiority of Bit-Sliced Interval Search Engine and Bit-Sliced Two-Way-Merge Sorter for our purposes.

3.1.2 Bit-Sliced Interval Search Engine

Table 3.2. Sort hardwares

h: number of comparators.
t: maximum sorting time.
D: maximum delay for the first output preparation after the last input.

bit-slicing

h	t	D		example
1	n^2	n^2	v	bubble sort
1	$n\log n$	$n\log n$		heap sort
$\log n$	n	$\log n$		pipeline merge sort (Todd 1978)
			v	(Tanaka 1984a)
$\log n$	n	0		pipeline heap sort (Tanaka et al. 1980)
n	n	n		parallel even-odd sort (Baudet and Stevenson 1978)
n	n	$\log n$		parallel tree sort (Bentley and Kung 1979)
n	n	0		parallel enumeration sort (Yasuura et al. 1982)
n	n	0		reboud sort (Chen et al. 1978)
n	n	0		pipeline bubble sort (Kung 1980)
n	$\sqrt{n}$	$\sqrt{n}$		mesh-connected bitonic sort (Thomson and Kung 1977, Nassimi and Sahni 1979)
n	$\log^2 n$	$\log^2 n$		Shuffle-connected bitonic sort (Stone 1971)
n^2	$\log n$	$\log n$		(Muller and Preparata 1975)
n	$\log n$	$\log n$		(Hirschberg 1978)
$n^{1+1/k}$	$k\log n$	$k\log n$		(Hirschberg 1978)
$n\log n$	$\log n$	$\log n$		(Preparata 1978)
$n^{1+1/k}$	$k\log n$	$k\log n$		(Preparata 1978)

An interval search engine (ISEE) performs batch search operations. It searches a same table of n keywords for different search keys. Let T(i) denote the i+1st keyword in this table T. The keywords are assumed to be arranged in a nondecreasing order. The table is stored in an engine preceding to the batch search processing, in which m search keys are sent to the engine one after another as a stream (k_0, k_1, ..., k_{m-1}). For each input key k, the ISEE outputs

an interval (A^L, A^R) of table addresses. These are the minimum addresses that satisfy respectively the following two conditions: $T(A^L) \geq k$ and $T(A^R) > k$. Their difference $A^R - A^L$ is equal to the number of keywords in T that are equal to k. In an ISEE, a search table is represented by a binary tree called a left-sided binary tree. An ISEE with L levels can store a table with no more than 2^L-1 keywords. At each level j of a tree, the number of nodes that are loaded with keywords is denoted by LOAD(j), and be referred to by the load factor of this level. An ISEE has a hardware configuration similar to a Search Engine (Tanaka et al. 1980) (Fig. 3.1). The logic circuits at each level j get a search key k and a pair of addresses $(w^L(j), w^R(j))$ from the upper level and outputs k and $(w^L(j+1), w^R(j+1))$ to the next level j+1. The values of $w^L(0)$ and $w^R(0)$ are assumed to be always zero. Let $T^j(i)$ denote the keyword stored in the node at the intra-level address i in the level j. The addresses $w^L(j+1)$ and $w^R(j+1)$ are calculated as follows:

$$w^L(j+1) = 2*w^L(j)+CO^L(j),$$

$$COND^L = k \leq T^j(w^L(j)) \text{ or } w^L(j) > LOAD(j)-1$$

$$CO^L(j) = \text{if } COND^L \text{ then } 0, \text{ else } 1,$$

$$w^R(j+1) = 2*w^R(j)+CO^R(j),$$

$$COND^R = k < T^j(w^R(j)) \text{ or } w^R(j) > LOAD(j)-1$$

$$CO^R(j) = \text{if } COND^R \text{ then } 0, \text{ else } 1.$$

The search result (A^L, A^R) for a search key k is obtained as the output addresses from the bottom level.

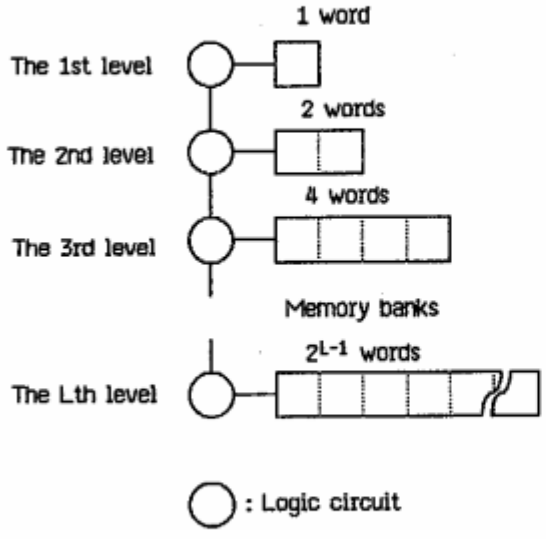


Fig. 3.1. ISEE hardware configuration.

Now, let us consider how to realize a bit-sliced architecture of an ISEE. First we shall modify an ISEE to have two bits of output signals $CO^L(j)$, $CO^R(j)$ at each level j. Such a modified ISEE is referred to by an MISEE.

A bit-sliced ISEE with n-bit wordlength is defined as a module that is connected to an MISEE with m-bit wordlength to form a new MISEE with (n+m)-bit wordlength. Each of its levels has two output lines $CO^L(j)$, $CO^R(j)$, and two input lines $CI^L(j)$, $CI^R(j)$. These two input lines $CI^L(j)$ and $CI^R(j)$ are respectively connected to $CO^L(j)$ and $CO^R(j)$ of the corresponding level of the preceding m-bit MISEE. An n-bit bit-sliced ISEE with each $CI^L(j)$ and $CI^R(j)$ respectively set to zero and one works in the same way as an n-bit MISEE. The bit-sliced ISEE can not see the boundaries $w^L(j)$ and $w^R(j)$ calculated at each level of its lefthand MISEE. However, we first assume they are visible. They are denoted by $W^L(j)$ and $W^R(j)$ for the distinction from those of the bit-sliced ISEE.

As to the calculation of $CO^*(j)$ (* denotes L or R), four possible cases should be considered:

(1) Case X^* : $W^L(j) = w^*(j) = W^R(j)$.
(2) Case L^* : $W^L(j) = w^*(j) < W^R(j)$.
(3) Case N^* : $W^L(j) < w^*(j) < W^R(j)$.
(4) Case R^* : $W^L(j) < w^*(j) = W^R(j)$.

These signals $CO^L(j)$ and $CO^R(j)$ are determied as shown in Table 3.3.

Table 3.3. Calculations of $CO^L(j)$ and $CO^R(j)$ and state transitions of a search process.
* denotes L or R.

Case	$CI^L(j)$	$CI^R(j)$	COND*	CO*(j)	Next State
X*	0	0	-	0	X*
	0	1	F	1	R*
	0	1	T	0	L*
	1	0	-	-	-
	1	1	-	1	X*
L*	0	-	F	1	N*
	0	-	T	0	L*
	1	-	-	1	L*
N*	-	-	F	1	N*
	-	-	T	0	N*
R*	-	0	-	0	R*
	-	1	F	1	R*
	-	1	T	0	N*

Although we have assumed that $W^L(j)$ and $W^R(j)$ are visible, actually they are not. This problem is solved by introducing states

of a search process. For each search key, its search process changes its state as it moves from the top level to the bottom level. At the top level, each search process initializes its state to (X^L, X^R). The state transition is specified by two automata, each of which specifies transitions among either the set $\{X^L, L^L, N^L, R^L\}$ or the set $\{X^R, L^R, N^R, R^R\}$. These automata are described in Table 3.3.

3.1.3 Bit-Sliced Two-Way-Merge Sorter

S. Todd (TODD 1978) proposed a sorting algorithm that repetitively applies merge operations to every two sorted runs in an input stream to increase the length of sorted runs. The initial input stream is considered as a sequence of sorted runs of length one. In order to perform these repetitive merge operaions in a pipeline fashion, a two-way-merge sorter has a hardware configuration similar to that of an ISEE. At every time when the next two input runs arrive at a stage, the logic circuit at each level begins to merge these two runs to output a merged run to the next level. A hardware module with L levels outputs from the bottom level a sorted run of length 2^L.

A bit-sliced architecture of this module can be easily designed if we can find out how to slice a merger used at each level of this module. Let L and R denote two sorted streams of same length, and L(i), R(i) their i+1st elements. The logic circuit at each level can be decomposed into two parts. The first part receives an input element at every step and stores it at a proper address of the memory bank at this level, while the second part merges two streams, whose next elements are always guaranteed to have been already stored in the memory bank by the first part of the circuits.

A bit-sliced merger has two 1-bit input lines LI and RI, and two 1-bit output lines LO and RO. These lines are used to connect multiple bit-sliced mergers to form a single merger. The operation of the i-th slice module is delayed (i-1) steps from that of the leftmost module. Let the left pointer lp point to the next element of the left stream, and rp point to that of the right stream. If L(lp) and R(rp) are equal at the leftmost slice, we will advance both of the two pointers, and make the module at this slice to output one value that is equal to both L(lp) and R(rp). The signal LO denotes the advance of the left pointer, while RO the advance of the right pointer. They are set to one if their corresponding pointers are advanced. Otherwise, they are set to zero.

Now let us consider the second slice of a merger. The second slice of a merger operates in a similar way as the leftmost one does unless it reaches either the left boundary or the right boundary of the leftmost slice. If it reaches, say, the left boundary, it must stop the advance of the left pointer. The following output must be selected from the right stream untill the right pointer also reaches the right boundary of the leftmost slice.

Let us first introduce several notations:

C : A counter that counts how many times the two pointers are simultaneously advanced. Initially zero.

v_0 : The previous output; initially zero.

D^L : The difference of lp between the current slice and the preceding slice. It becomes zero when the module reaches the left boundary of the preceding slice. Otherwise, it is kept positive. Initially zero.

D^R : Similarly defined except that 'left' is replaced with 'right'.

$D^{L\prime} = D^L + LI$.

$D^{R\prime} = D^R + RI$.

The classification and the operations in each case are described below:

Case 1. $D^{L\prime} = D^{R\prime} = C = 0$: nonexistent.

Case 2. $D^{L\prime} = D^{R\prime} = 0$, $C\ (= n) > 0$

$(LO, RO) \leftarrow (0, 0)$; output v_0; $C \leftarrow C-1$;

$D^L \leftarrow D^{L\prime}$; $D^R \leftarrow D^{R\prime}$.

Case 3. $D^{L\prime} = 0$, $D^{R\prime} > 0$

Case 3.1. $C = 0$ or $R(rp) = v_0$

$(LO, RO) \leftarrow (0, 1)$; output $R(rp)$;

$v_0 \leftarrow R(rp)$; $rp \leftarrow rp+1$; $D^L \leftarrow D^{L\prime}$;

$D^R \leftarrow D^{R\prime}-1$.

Case 3.2. $C \neq 0$ and $R(rp) \neq v_0$

$(LO, RO) \leftarrow (0, 0)$; output v_0; $C \leftarrow C-1$;

$D^L \leftarrow D^{L\prime}$; $D^R \leftarrow D^{R\prime}$.

Case 4. $D^{L\prime} > 0$, $D^{R\prime} = 0$

Case 4.1. $C = 0$ or $L(lp) = v_0$

$(LO, RO) \leftarrow (1, 0)$; output $L(lp)$;

$v_0 \leftarrow L(lp)$; $lp \leftarrow lp+1$; $D^L \leftarrow D^{L\prime}-1$;

$D^R \leftarrow D^{R\prime}$.

Case 4.2. $C \neq 0$ and $L(lp) \neq v_0$

(LO, RO) ← (0, 0); output v_0; C ← C-1;

$D^L \leftarrow D^{L_t}$; $D^R \leftarrow D^{R_t}$.

Case 5. $D^{L_t} > 0$, $D^{R_t} > 0$

Case 5.1. min(L(lp), R(rp)) ≠ v_0 and C ≠ 0

(LO, RO) ← (0, 0); output v_0; C ← C-1;

$D^L \leftarrow D^{L_t}$; $D^R \leftarrow D^{R_t}$.

Case 5.2. C = 0 or min(L(lp), R(rp)) = v_0

Case 5.2.1. L(lp) < R(rp)

(LO, RO) ← (1, 0); output L(lp);

v_0 ← L(lp); lp ← lp+1; $D^L \leftarrow D^{L_t}$-1;

$D^R \leftarrow D^{R_t}$.

Case 5.2.2. L(lp) > R(rp)

(LO, RO) ← (0, 1); output R(rp);

v_0 ← R(rp); rp ← rp+1; $D^L \leftarrow D^{L_t}$;

$D^R \leftarrow D^{R_t}$-1.

Case 5.2.3. L(lp) = R(rp)

(LO,RO) ← (1,1); output L(lp);

v_0 ← L(lp); C ← C+1; lp ← lp+1;

rp ← rp+1; $D^L \leftarrow D^{L_t}$-1; $D^R \leftarrow D^{R_t}$-1.

The slicing method described above is applicable to a two-way-merge sorter.

The two different bit-sliced VLSI architectures both require (4L+1) pins per chip. For L=12 (i.e., capacity = 4095 words), their pin complexity becomes 49+α, where α pins are necessary for power supply, clock supply, and mode control. This number seems to be acceptable. The bit-sliced ISEE with L levels and 1 bit width consists of 2^L-1 memory cells (, for L=12, 4095 cells) as a whole, and two 4 state automata at each level, while the bit-sliced sorter with L levels and 1 bit width consists of $3(2^L-1)$ memory cells as a whole, and a simple logic circuit with several registers at each level. Therefore, even if we use static RAM technology, each module with L levels and 1 bit width requires less than 10^5 transistors. This number ensures their feasibility.

3.2 Segment Processor

Segment processor has an architecture consisting of a high performance microprocessor with more than 4 page storage space besides its progrm space, and two engines, respectively for batch search and sort, that have enough capacity to process two pages of data. These engines work as subprocessors of the main microprocessor and they are capable of block data transfer to and from the main processor memory.

Page operation commands that segment processors receive and execute have either of the following formats;

PA^* ← BO ($Q_1(PA_1)$, $Q_2(PA_2)$),
(Q_i may be nil.)

PA^* ← UO (Q(PA)),
(either UO or Q may be nil.)

$PA_1{}^*$ ← condense($PA_2{}^*$),

where

PA : a page address allocated to a single page variable.

PA^* : a page address allocated to a single page variable or to a multiple page variable.(If it has been allocated to a multiple page variable as the first page, it is marked with a multiple page indicator ◇.)

Q_i : selection and/or restriction operations.

BO : a binary operation such as intersection, union, difference, join, and division.

UO : a unary operation such as projection, sort, count, sum, average, maximum, and minimum.

A command with the first format requests execution of a binary operation. The segment processor reads out each source operand page from Shared Page Buffer. During the page transfer, it selects only those tuples that satisfy the qualification condition of Q_i and stores them in its local memory. It performs the binary operation using dedicated engines, and write down the result in the destination operand page. If the destination operand has a multiple page indicator, the segment processor requests Page Buffer Manager for one more page whenever it requires more pages to save the result. Page Buffer Manager returns page allocation information to the requesting segment processor. When a segment processor completes the execution, it sends a completion status code to Control Subsystem with the list of multiple destination pages if any.

A command with the second format requests unary operation execution on a single page. The segment processor reads out the page from Shared Page Buffer and selects tuples satisfying the qualification condition of Q during the page transfer. Then it

performs the unary operation. The result is saved in the same way as mentioned above.

The third format command requests a special operation. Pages are not always fully loaded. It is sometimes desirable to condense a set of pages into a smaller set of pages. It will decrease page references. The third format command requests this operation. It is the only command that allows a multiple-page operand in a source operand position. The source page list is sent to the segment processor via one page of Shared Page Buffer. The segment processor first reads out this page. It repeatedly reads out the pages in this list, condenses them and saves the result in the same way as other format commands save their result.

Each segment processor begins to search for a next task whenever it finishes a task. It continues to probe Page Operation Command/Status Network untill it gets a task.

3.3 Shared Page Buffer

As will be explained in the next section, our segmentation schemes and our concurrency control based on them can generate a large amount of concurrently executable segment processing tasks. If databases are stored in an ideal shared storage device that allows sufficiently many concurrent accesses from segment processors, the assignment of each segment processing task to a different segment processor realizes massively parallel database processing. In practice, databases are divided and stored in multiple disk units, any of which can not be concurrently accessed by multiple segment processors. Besides, disk units require seriously large access time.

This problem was solved in my recent paper (Tanaka 1984b). The multiport page-memory architecture proposed in it provides a new type of a shared storage space. for massively parallel processing. It can afford 10^3~10^4 ports each of which can read or write an arbitrary page in the shared storage space without causing any conflict nor any wait. The principal idea is based on the fact that the access sequence of words in a page may be arbitrary.

Its basic architecture with n ports consists of n equal-sized memory banks and an n×n interconnection network with its controller (Fig. 3.2). The n memory banks as a whole forms a memory space divided into equal-sized pages of consecutive words. The page size S is assumed to be a multiple of n. Consecutive words in each page are arranged horizontally across all memory banks. One horizontal line of this arrangement is called a track. Consequently, a track has n words. A word address in a page buffer is denoted by a pair (p, d); p is a page address and d is a displacement in that page. In orthogonal arrangement in n memory banks, the word with address (p, d) locates in the memory bank M(d mod n), and its address in this bank is $\lfloor (pS+d)/n \rfloor$. If the network in Fig. 3.2 is capable of connecting each port i and the memory bank M(α(i)) for a permutation α of n integers from 0 to n-1, each port can access a single word in an arbitrary page concurrently with other ports. The displacement d of this word in each page should be kn+α(i) with k being one of the integers between 0 and S/n-1.

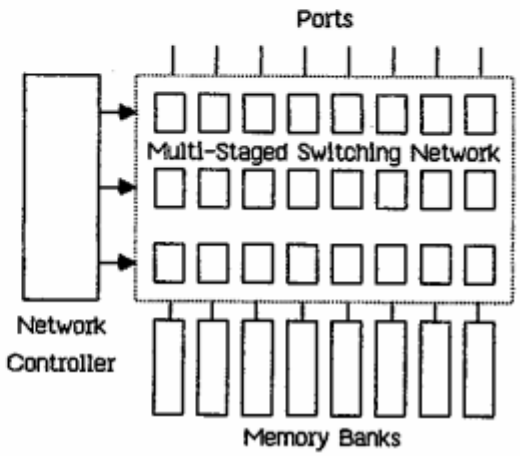


Fig. 3.2. Shared Page Buffer Architecture.

A scan sequence of order n is defined as follows.

Definition 3.1.
A permutation sequence $\alpha_0, \alpha_1, \ldots$ is a scan sequence of order n if it satisfies both of the following conditions:
(1) Each α_i is a permutation of n integers from 0 to n-1.
(2) For each integer i between 0 and n-1, and any nonnegative integer j, the sequence $(\alpha_j(i), \alpha_{j+1}(i), \ldots, \alpha_{j+n-1}(i))$ is a permutation of n integers from 0 to n-1.

A network is said to realize a permutation α if it can connect each port i to the memory bank α(i). A permutation realized by a network N at time tΔ, where Δ is a cycle time, is denoted by N_t. If the sequence $N_0, N_1, \ldots$ is a scan sequence, the network N is called a scan network. A scan network has the following property.

Theorem 3.1
The configuration in Fig. 3.2, when a scan network is used, enables each port to start reading or writing any page at any time independently from the other ports' operation. The operation takes no more consecutive memory cycles than the page size.

proof
Assume that a port i begins to access a page p at time t. In a page access mode, words in a page need not be accessed in the ascending order of their displacements. Let us consider an access sequence;

$$(p,N_t(i)),\ (p,N_{t+1}(i)),\ \ldots\ ,\ (p,N_{t+n-1}(i)),$$

$$(p,n+N_{t+n}(i)),\quad \ldots\quad ,\ (p,n+N_{t+2n-1}(i)),$$

$$\ldots$$

$$\ldots\quad (p,kn+N_{t+kn+j}(i)),\quad \ldots$$

$$\ldots\quad ,\ (p,S-n+N_{t+S-1}(i)).$$

From the definition of a scan network, this sequence accesses at each time a memory bank that is free from accesses by the other ports. Since, for any k, the sequence $(N_{t+kn}(i),\ N_{t+kn+1}(i),\ \ldots\ ,\ N_{t+(k+1)n-1}(i))$ is a permutation of n integers from 0 to n-1, the set of displacements

$$kn+N_{t+kn}(i),\ kn+N_{t+kn+1}(i),\ \ldots\ ,$$

$$kn+N_{t+(k+1)n-1}(i)$$

covers all the displacements between kn and (k+1)n-1. This implies that the displacement set in the access sequence above covers all the displacements between 0 and n-1. Besides, the number of addresses in the sequence is equal to the page size. Therefore, the theorem holds true. □

The access sequence mentioned in the above proof is called a standard access sequence of a scan network. Its j-th address is represented as

$$(p,\ \lfloor j/n \rfloor n+N_{t+j}(i)).$$

A scan network that repeatedly realizes n different permutations $\alpha_0,\ \alpha_1,\ \ldots\ ,\ \alpha_{n-1}$ in this order is called a periodic scan network. Generally speaking, the periodicity decreases the network hardware complexity. As shown in Fig. 3.3, periodic scan networks can be constructed from $\log_2 n$ rows of basic switching modules and a controller that changes the state of each switching module. This is similar to the case of n×n interconnection networks studied to allow flexible connections between processors and memory banks (Lawrie 1975, Pease 1977, Goke and Lipovski 1973, Parker 1980). Different from them, a periodic scan network need not allow arbitrary connections between two groups. For simplicity, all of the switching modules at each row are assumed to be controlled by a single control signal. The controller needs to provide only $\log_2 n$ signals. An array of $\log_2 n$ control signals at each instance can be represented by a $\log_2 n$ bit binary number with the top level signal as its MSB and the bottom as its LSB. This number is called a control vector. The j+1st control vector is denoted by c(j). A special sequence of control vectors satisfying,

$$\text{for any } i,\ \ c(i) = i \bmod n$$

is called the regular sequence of control vectors. In the sequel, every control vector sequence is regarded as regular. Among periodic scan networks, the following two types are worth mentioning. The number of ports is assumed to be a power of two.

1. A rotary network
   $N_j(i) = i+j \bmod n$

2. A shuffle scan network
   $N_j(i) = i \oplus (j \bmod n)$,
   where $i \oplus j$ is a bitwise exclusive OR of i and j.

A rotary network uses 1×2 switch modules, and has a connection pattern as shown in Fig. 3.3. The controller repeatedly sends a sequence of control vectors 0, 1, 2, changes its partner bank incrementally from i to n-1, and then from 0 to i-1. A shuffle scan network uses 2×2 switch modules, and has a different connection pattern that is well known as omega network (Lawrie 1975). Its controller also sends a control vector sequence 0, 1, 2, ... , n-1 in this order. The j-th partner of a port i is determined by the bitwise exclusive OR of i and (j-1 mod n). For n=8, port 0, for example, repeatedly changes its partner bank in this order 0, 1, 2, 3, 4, 5, 6, 7, while port 3 changes its partner in a different order as 3(=011⊕000), 2(=011⊕001), 1(=011⊕010), 0(=011⊕011),

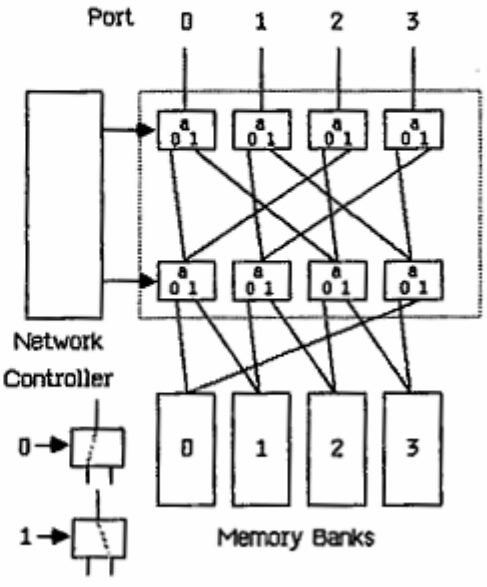


Fig. 3.3. 4x4 rotary scan network.

7(=011⊕100), 6(=011⊕101), 5(=011⊕110), 4(=011⊕111).

Memory interleaving can be applied to improve access rate in consecutive retrievals.

In the preceding paragraphs, we neglected the delay of basic switch elements used as scan network components. Actually, an access from a port to one of the memory banks needs to pass through $\log_2 n$ switches in an n×n network. If n is 1000~10000, the number of switches to pass through becomes as large as 10~13. The additional delay caused by these switches becomes comparable with the memory access time. This almost doubles the effective memory access time. This problem is resolved by pipelining as many accesses as the number of rows in the network.

Each port of a multiport page buffer has a constant access rate. Some devices can not transfer data without fluctuations occuring in transfer rates. However, the consecutive accesses to the shared page buffer does not allow access rate fluctuations. Besides, some devices such as disk units can not start a page read/write operation arbitrarily. Some other devices can not access words on a page in an arbitrary order. These devices can not be directly connected to the shared buffer.

These problems are solved by introducing two n word track buffers inbetween such devices and the shared page buffer ports. Two track buffers are used alternately. While one of them is receiving the next n words from either the device or the shared buffer, the other track buffer is sending the last received n words to the other system. Since the transfer rate of the shared page buffer is much faster than the transfer rate of the device, the transfer of a page to and from the shared buffer has to wait for one of the track buffers to be filled up or be emptied after every n word transfer. If the buffers are considered as a part of such a device, the delay introduced by this interface is no more than n clock cycles, which is independent from the page size and much smaller than the transfer time for a single page.

4 CONTROL SUBSYSTEM

4.1 File Structure

4.1.1 Colored Binary Trie

File segmentation is inevitable to cope with large files of information even in the design of database machines if we want to enlarge their capacity. It divides database processing into two levels, i.e., search for segments necessary for transaction processing, and relational operation processing on each segment or each segment pair. In the MPDC architecture, it separates macroparallelism and microparallelism. If files are segmented arbitrarily, most queries require accesses to all the segments, which severely abates the system performance.

File segmentation schemes are the clustering techniques that appropriately distribute file records to a large set of segments so as to balance and minimize the number of segment accesses necessary to answer various queries. Every segmentation scheme consists of two components, a directory and a set of segments. A directory is a set of rules that specifies how to distribute the file records to a set of segments. It may be represented by a hash function, a table, or a search tree. Every segment has the same finite size as the page size, and hence it may possibly overflow.

For the retrievals based on the values of a single key attribute, whether it is primary or not, a lot of segmentation schemes have been proposed. Some of them have been practically used and approved. However, segmentation for the retrievals based on the values of multiple secondary key attributes has not been much explored yet, except extended k-D tree (Chang and Fu 1981) and k-D trie (Orenstein 1982).

Our studies on this problem proposed two colored binary schemes (Tanaka 1983b). Here, these schemes will be briefly reviewed. Suppose first that we have a relational file of records each containing n secondary keys, where each secondary key has a fairly large number of possible values. We can map the records whose secondary keys are $(k_0, k_1, \ldots, k_{n-1})$ to the (n*m)-bit number

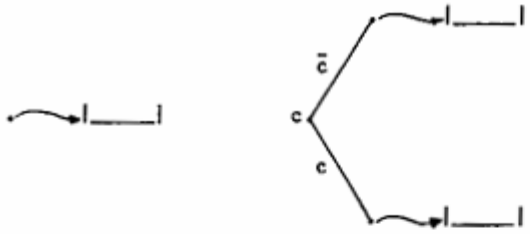
$$h_0(k_0)h_1(k_1)\ldots h_{n-1}(k_{n-1}),$$

where each h_i is a hash function that maps the values of the (i+1)st secondary key attribute into a set of m-bit values. Use of partially order preserving hash functions (Tanaka 1983a) is desirable for those attributes that possibly appear in a range search condition. The above expression stands for the juxtaposition of n m-bit values.

Now the segmentation of a relational file can be stated in an abstract manner as follows. Suppose that we have a lot of beeds each colored with one of the different colors, $c_0, c_1, \ldots, c_{n-1}$. The set of these colors is denoted by C. A bead with c_i color is referred to as a c_i-bead. Each bead is labeled with an m-bit value. There may be beads with a same color and a same label. A rosary is a string of n beads each having a different color. The c-label of a rosary is defined as the label on its c-bead.

Rosaries are made one by one, choosing

an arbitrary label for each color. They are stored in a set of drawers each having a constant capacity. Initially, only a single drawer is used to store rosaries, and hence its directory has only one entry (Fig. 4.1 (a)). If an overflow occurs, the rosaries in the drawer should be divided into two classes. They can be divided based upon the values of a certain bit of a certain color label. For this division, we use the most significant bit of some color label. The directory will come to have two entries corresponding to two new drawers that store the two classes. It can be represented as a binary trie with two leaves and a root that is painted with the color whose labels were used as a basis of the division (Fig. 4.1 (b)). If one of the two drawers overflows again, its contents are further divided into two classes. In general, the division of a cluster can be based upon an arbitrary bit of an arbitrary color label unless this bit has been already used as a basis of another division in the process of having produced this cluster. We use, in every division, the leftmost unused bit of some color label. The directory of drawers that describes the rules of cluster division can be represented as a colored binary trie. It is a binary trie whose internal nodes are painted with one of the n colors.



directory drawer directory drawer

(a) the initial state (b) after the split by a color c

Fig. 4.1 The division of the contents of an overflowing drawer based on a bit of the c-labels of rosaries

In a colored binary trie, the left branch from a node colored with c is represented by $\bar{c}$, while the right by c. The concatenation of the representation of branches along the path from the root to any other node uniquely identifies that node in the trie. This identifier is referred to as a node code. For a node code α and each color c, we define the c-code of that node as a bit sequence that is obtained by first deleting all the appearances of c' and $\bar{c}'$ from α for each c' different from c, and then replacing c and $\bar{c}$ respectively with '1' and '0'. The c-code of the node with a node code α is denoted by $c(\alpha)$, while the length of α and that of $c(\alpha)$ are respectively represented by $\rho(\alpha)$ and $\rho(c(\alpha))$. The node α of a colored binary trie stands for the cluster of rosaries whose c-labels begin with $c(\alpha)$ for each color c.

Each customer requests a search for all those rosaries with a specified label on a specified color bead. The processing of such a request first requires a search of a directory for drawers that possibly contains some of the requested rosaries. Then it requires searches of these drawers for all the rosaries of the requested type. The wait time of a customer is approximately proportional to the number of drawers to be searched. Fig. 4.2 shows an example directory represented by a colored binary trie with three colors, R, G, and B. Segments are denoted by the leaf nodes. They are labeled with the numbering from 1 to 6. Let the search for rosaries with the c-label v be referred to as a 'c=v' search. For the search of R=00...00, it is necessary to pull out three drawers 1, 2, and 3. For R=00...01, the same set of drawers is required. Generally, these drawers are necessary and sufficient to search for all the rosaries with the R-labels beginning with 00. These search requests are represented by R=00**...*, where '*' stands for an arbitrary binary value. A search request B=0**...* requires to pull out four drawers, 1, 2, 4, and 5. The number of necessary drawers varies depending on the color c and its label v. This number is denoted by naccess(T, c, v), where T denotes a directory trie.

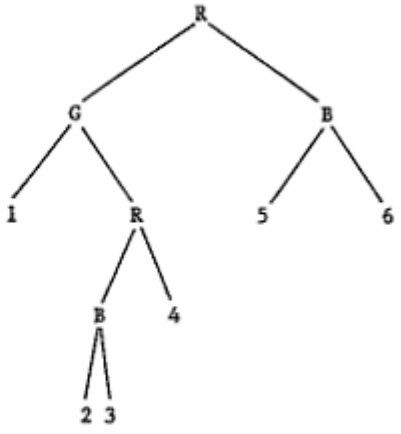


Fig. 4.2 An example directory represented by a colored binary trie with 3 colors

Let Cavg(T, c) and Cworst(T, c) respectively denote the average and the maximum number of segment accesses necessary for searches based on the values of the c-label, i.e.,

$$\text{Cavg}(T, c) = \underset{v \in \{0,1\}^m}{\text{average}} (\text{naccess}(T, c, v))$$

$$Cworst(T, c) = \max_{v\in\{0,1\}^m} (naccess(T, c, v)).$$

Two kinds of access costs can be defined:

1. average cost

$$cost^1(T) = \text{average}_{c\in C} (Cavg(T, c)).$$

2. worst cost

$$cost^2(T) = \max_{c\in C} (Cworst(T, c)).$$

Suppose that we have a directory T and that one of its drawers overflows. We want to choose the most desirable color to split the overflowing leaf of T so that the result trie may have the least cost. Suppose that the overflow occurs at a leaf with a node code α. Let a trie obtained by splitting this leaf based on the leftmost unused bit of c-label be denoted by $new(T, \alpha, c)$. The most desirable color is formally defined as the one that minimizes the following function of the color variable c;

$$Ccost^i_{T,\alpha}(c) = cost^i(new(T, \alpha, c)).$$

There can be two different schemes corresponding to the two cost functions. The best average scheme minimizes $Ccost^1_{T,\alpha}(c)$, while the best worst scheme minimizes $Ccost^2_{T,\alpha}(c)$. The best average scheme results in a good performance throughput, while the best worst scheme improves responce time.

4.1.2 Best average scheme

For a colored trie T and an overflowing leaf α, $Ccost^1_{T,\alpha}(c)$ is calculated as follows.

Theorem 4.1 (Tanaka 1983b)

$$Ccost^1_{T,\alpha}(c) = cost^1(T)+(1/n)\ \text{sum}\ (1/2)^{\rho(c'(\alpha))} - (1/n)(1/2)^{\rho(c(\alpha))}. \qquad (4.1.1)$$

This theorem says that, in the best average scheme, the split of a leaf with a node code α should choose a color that minimizes $\rho(c(\alpha))$. Suppose that n colors are $c_0, c_1, \cdots,$ and c_{n-1}. As a special case of the best average schemes, a scheme is a best average scheme if it selects, for the splitting of a node at the i-th level, the color c_j whose suffix j is congruent to i-1 modulo n. Such a scheme is called a regular best average scheme. Regular best average schemes result in the same schemes as k-D tries. Actually, k-D trie is a special implementation of the best average colored binary trie scheme.

4.1.3 Best worst scheme

When an overflow occurs at some leaf of a colored trie, the best worst scheme splits this leaf by such a color that minimizes $Ccost^2_{T,\alpha}(c)$. If both c' and c'' are different from c then $Cworst(new(T, \alpha, c'), c)$ is equal to $Cworst(new(T, \alpha, c''), c)$. Let c^+ denote a representative of the colors that are different from c. Then the following theorem holds.

Theorem 4.2 (Tanaka 1983b)
If a color c maximizes $Cworst(new(T, \alpha, c^+), c)$ then it minimizes $Ccost^2_{T,\alpha}(c)$.

Let $L_c(v)$ and $W_c(v)$ be defined as follows;

$$L_c(v) = card(\{\alpha \mid \alpha\in T,\ c(\alpha)=v\}),$$

$$W_c(v) = \text{if, for any } \alpha\in T,\ c(\alpha)\neq v \text{ then } 0 \text{ else } L_c(v)+\max(W_c(v\circ 0), W_c(v\circ 1)),$$

where card(S) denotes the cardinality of a set S.

Theorem 4.3 (Tanaka 1983b)

$$Cworst(T, c) = W_c(\epsilon).$$

Let us define $W^\alpha_c(v)$ as follows;

$$W^\alpha_c(v) = \begin{cases} W_c(v)+1 & \text{if } v=c(\alpha) \\ W_c(v) & \text{elseif } v\not\in P(c(\alpha)) \\ \text{for } b\in\{0,1\} \text{ such that } v\circ b\in P(c(\alpha)) & \\ \quad W_c(v) & \text{if } W_c(v\circ\bar{b})\geq W_c(v\circ b)+1 \\ \quad L_c(v)+W^\alpha_c(v\circ b) & \text{else} \end{cases} \qquad (4.1.2)$$

where P(v) denotes a set of prefixes of a finite binary sequence v. Then the following theorem holds.

Theorem 4.4 (Tanaka 1983b)

$$Cworst(new(T, \alpha, c^+), c) = W^\alpha_c(\epsilon).$$

The algorithm for the best worst scheme is stated as follows, where a finite set S_c is defined as

$$S_c = \{c(\alpha) \mid \alpha \text{ is a leaf of } T\}.$$

Algorithm

1. Compute $W^\alpha_c(\epsilon)$ for each c.

The number of steps necessary for the computation of $W^\alpha_c(v)$ is proportional to the length of $c(\alpha)$. Therefore, the total number of steps necessary to compute $W_c(v)$ for n different colors is proportional to

$$\sum_{c} \rho(c(\alpha)) = \rho(\alpha),$$

which is bounded by the height of the colored trie.

2. Choose a color c that maximizes Cworst(new(T, α, c^+), c).

Since Cworst(new(T, α, c^+), c) is $W_c^{\alpha}(\epsilon)$, what we have to do is to find out a color that maximizes $W_c^{\alpha}(\epsilon)$. If there are more than one candidate, choose one that minimizes $\rho(c(\alpha))$.

3. Split the overflowing node by the selected color c_0, and update $L_c(v)$ and $W_c(v)$ for each c and $v \in P(c(\alpha))$.

For any c different from c_0, and any $v \in P(c(\alpha))$,

$$L_c^{new}(v) \leftarrow L_c^{\alpha}(v),$$

$$W_c^{new}(v) \leftarrow W_c^{\alpha}(v).$$

For $c=c_0$,

$$S_c^{new} \leftarrow (S_c - \{c(\alpha)\}) \cup \{c(\alpha)\circ 0,\ c(\alpha)\circ 1\},$$

$$L_c^{new}(v) \leftarrow \text{if } v=c(\alpha) \text{ then } L_c(v)-1$$
$$\text{elseif } v=c(\alpha)\circ 0 \text{ or } v=c(\alpha)\circ 1$$
$$\text{then } L_c(v)+1$$
$$\text{else } L_c(v).$$

$$W_c^{new}(v) \leftarrow \text{if } v \in S_c^{new} \circ \{0,1\} \text{ then } 0$$
$$\text{elseif } v \notin P(c(\alpha))$$
$$\text{then } W_c(v)$$
$$\text{else } L_c^{new}(v)$$
$$+\max(W_c^{new}(v\circ 0),$$
$$W_c^{new}(v\circ 1)).$$

Because of the same reason, the number of steps necessary for the update of $L_c(v)$ and $W_c(v)$ is proportional to the length of $c(\alpha)$. Therefore, the total number of steps necessary to update $L_c(v)$ and $W_c(v)$ for all different colors is proportional to $\rho(\alpha)$, which is bounded by the height of the colored trie.

Our segmentation schemes have the following advantageous features.

1. It is completely adaptive, and has no restrictions on the number of segments and of attributes.
2. It can be arbitrarily chosen either to minimize the average number of segment accesses or to improve the worst case performance. This property is different from an extended k-d tree scheme and from a k-d trie scheme, which can minimize only the average. Besides, the minimization in them is performed under the restriction that the node splitting at each level uses a same secondary key. Our new scheme assumes no such restrictions.
3. A search of the directory with N segments and its local rewriting need only O(log N) time on an average for large N. Especially, if the values of the secondary keys are independently and uniformly distributed, these operations need no more than O(log N) time for large N.
4. The regular best average scheme makes the average number of segment accesses necessary for the processing of a relational selection operation no more than $O(N^{(n-1)/n})$, where N and n are respectively the number of relational records and the number of secondary key attributes. On the other hand, it is proved that, if the record values are uniformly distributed, no segmentation scheme can make this file access cost less than $O(N^{(n-1)/n})$, whether it is static or adaptive.

The computer simulations have shown various desirable features of these schemes. Among them, the following features are worth mentioning.

1. The loading factor is about 70 %, which is fairly good.
2. In the best average scheme, the expected number of segment acesses necessary for the processing of a relational selection operation almost coincides with the lower bound of the average cost, and it is almost independent from the distribution of record values. This is shown in Fig. 4.3.
3. In the best worst scheme, the responce

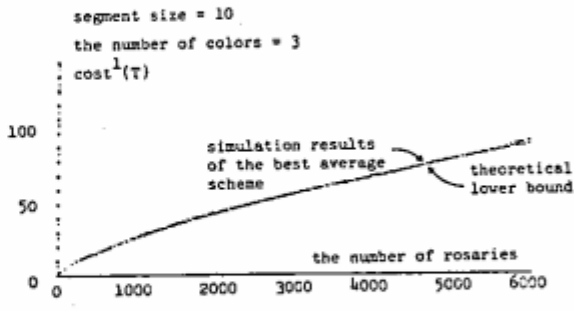


Fig. 4.3. Experimental analysys of colored binary trie schemes.
(a) simulated average number of segment accesses in best average scheme together with its theoretical lower bound. (The loading factor is assumed to be 70% in the computation of the theoretical lower bound.)

time of the processing of a relational selection operation is almost independent from the distribution of record values. Besides, the maximum number of segment accesses becomes very close to the expected number of segment accesses. In other words, the best worst scheme results in very small variance of the number of necessary segment accesses in the processing of various selection queries. This is shown in Fig. 4.4.

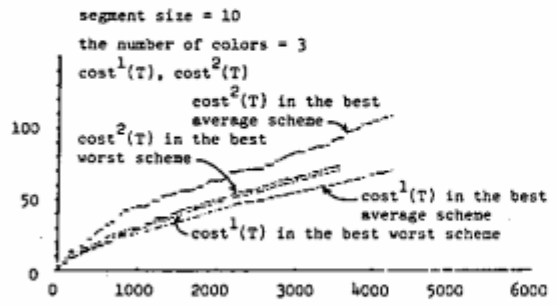


Fig. 4.4. Experimental analysys of colored binary trie schemes.
(b) comparison of the two schemes, i.e., the best average scheme and the best worst scheme, in the case in which record values are uniformly distributed.

4. In the proposed schemes, the number of segment accesses necessary for the processing of a relational restriction operation is approximately the same as in the case of a selection operation.
5. In the proposed schemes, any full equi-join of two relations each of which has O(N) segments and n secondary key attributes requires no more than $O(N^{2-(1/n)})$ joins of two segments. Otherwise, its maximum time complexity is $O(N^2)$. Besides, in our schemes, if $O(N^{(n-1)/n})$ size buffer is provided, any full equi-join of them requires no more than O(N) disk accesses. Otherwise, O(N) size buffer is necessary to achieve O(N) access complexity. In MPDC, if $O(^{(n-1)/n})$ segment processors and $O(N^{(n-1)/n})$ pages in Shared Page Buffer are available, any full equi-join of them requires no more than O(N) time. No database machines other than MPDC have ever achieved theoretically proved O(N) time processing of any full equi-join.

All of these desirable features of our schemes shows their applicability to the practical relational files and also to the large relational database machines.

4.2 Concurrency Control

4.2.1 Unified Approach to File Organization, Version Control, and Concurrency Control

Massive macroparallelism requires highly concurrent execution of segment read and segment write operations. Besides, high reliability requires a sound recovery mechanism that does not seriously lower system performance during its execution. The multiversion model of databases simultaneously satisfies these requirements. Existence of old versions enables us to bring the system back to its old state before a failed update operation.

A file is not necessarily provided with a new version of the whole file whenever it is updated. Otherwise, multiversion database systems are impractical. In segmented file organizations, it is sufficient to provide new segment versions only for modified segments. If files are clustered into segments to increase access locality, the number of modified segments in each update operation does not become large. Each segment can be revised independently. Version control of segments requires directory handling and concurrency control of segment processing. Therefore, these three functions should be integrated into unified control mechanism. Since MPDC uses colored binary trie schemes as its segmentation schemes, this section will give a multiversion concurrency control mechanism for colored binary trie schemes. Actually, the concurrency control described in this section is applicable to databases with a tree structured directory, which is not necessrily a colored binary trie scheme. Therefore, it will be formalized in its most general form. This section may be also applied to implementation of high-performance and high-reliability database management systems.

4.2.2 Models of Transactions and File Organization

Transactions are classified into two categories, i.e., read transactions and write transactions. Transactions with no update operations are read transactions, while others are write transactions. A file is considered as a tree structured set of objects. The root object of this tree may be interpreted as a relation directory that stores, for each relation, the physical location of its segment directory. A son of the root may be interpreted as the entry of the corresponding segment directory. A subtree with a root's son as its root corresponds to a hierarchically organized segment directory of a relation. A colored binary trie is an example. Each leaf node represents a segment, or more precisely, a segment address. In a multiversion directory, every node is allowed to have arbitrary number versions of the corresponding object. Therefore, each segment may have arbitrary number of segment versions. For simplicity, in other sections,

segment versions are simply called segments. Versions should be preserved while they are possibly referred to. Unnecessary versions should be deleted to decrease total number of storage segments.

Objects are lock units. Old versions for each object allow read transactions to read them while a write transaction is producing a new version of the same object. An object is modeled as a finite sequence of values of a same type, i.e.,

$$O = (v_0, v_1, \ldots, v_{n-1}),$$

where n is the number of versions of O and be denoted by n(O). A value v_i is called the i+1st version of the object O, and be denoted by O(i). The index i is called the version number of v_i. We call the first version O(0) the new version, O(1) the current version, and every remaining version an old version. Every new version is usually nil. It takes a non-nil value only while the object is being modified by a write transaction. When the update ends succesfully with a commit command having been issued by the transaction, the object is modified as

$$O \leftarrow (\text{nil}, O(0), O(1), \ldots, O(n(0))). \qquad (4.2.1)$$

This operation changes the version number of each version. Actually, versions that will not be further refered to are deleted and the remaining versions are compressed during this assignment. For the present, however, we assume that the number of versions is allowed to increase monotonically. This simplified model will be modified later in this section.

Associated with a database is a set of assertions called integrity constraints. A database is consistent if the current values of objects satisfy the given integrity constraints. We assume that a database is in correct state if it is consistent. A correct transaction, if executed alone, transforms the database from a correct state to another correct state. During intermediate states, even a correct transaction may violates the integrity constraints through its execution. Therefore, the concurrent execution of a set of correct transactions may produce an overall result that is not correct because of the interference of each transaction with another one. A set of correct transactions is considered to produce a correct result if they are executed without any intervention, or in other words, if they are executed one after another. Therefore, the correct execution of the transactions $T_1, T_2, \ldots, T_n$ will be correct if it produces the same effect as some arbitrary serial execution $(T_{P(1)}, T_{P(2)}, \ldots, T_{P(n)})$, where P is a permutation function. This condition is referred to as serializability. To achieve the correct execution of the concurrent transactions, they must be synchronized in some way. Usually, this is managed by various locking protocols. Here, we will propose a locking protocol for the above described new model.

Our protocol provides six kinds of lock operations for object locking. They are r-lock (read lock), w-lock (write lock), r-unlock (read unlock), w-unlock (write unlock), commit (commit operation), and roll-back (roll-back operation). Although lock operation names are similar to those in well-known theories, their semantics are quite different.

A read lock, when it is set on an object, keeps its current version value at this time, and ensures the readability of this value untill the lock is released by a corresponding read unlock operation. This current version value may change its version number during the execution, i.e., it may become an old version.

A write lock enables following update operations in the same transaction to exclusively posses the new version while they produce a new value on it. A write lock also ensures the readability of the current version value and prohibits its revision. If all update operations in a transaction have finished normally, a commit command is used to release all write locks. Each new version value produced by the transaction replaces the old current version value if the update operations have actually changed this new version value from nil. In this case, all values of an updated object O will be shifted to the past by the assignment statement (4.2.1). If no actual update has been done on this object, i.e., if O(0) remains nil, then no operation is performed. If the update operations have not finished normally, write unlock commands are used to release write locks. A single roll-back command can also nullify all update operations in the failed transaction. In these case, versions will not be revised and modified new versions are reset to nil.

In our protocol, neither w-lock, w-unlock nor roll-back is allowed to use in read transactions, while neither r-lock nor r-unlock is allowed in write transactions. This restriction does not reduce concurrency, because multiversion systems allow simultaneous setting of r-locks and a w-lock on a shared object. Each write transaction is allowed to issue no more than one commit command.

Transactions are assumed to have their identification number. For each object O, the set of transactions that have set a lock on the i+1st version of O is denoted by L(O(i)). The set L(O(0)) is either empty or a sigleton with one write transaction number. For each i, L(O(i)) is either empty or a set

with only read transactions.

Each object version represents an object value during a certain time period. To clarify this time period, it is necessary to introduce a logical clock LC. It is a counter that is initially reset to zero and has sufficiently many bits. It is incremented by one whenever a write transaction issues a commit command. The value of LC defines a nonlinear monotonic function of the actual time. It will be called the logical time. A new version value becomes referable, i.e., its version number becomes positive, when the modifying transaction issues a commit command. Therefore, the time stamp of a version is defined as the logical time when this version value became referable, i.e., when a commit command made it a current version. For $i \geq 1$, let ts(O(i)) denote the time stamp of a version O(i). In logical time, a version value of O(i) was the current value of the object O during a time period $[ts(O(i)), ts(O(i-1)))$, where $[t_1, t_2)$ denotes a set of real numbers that are greater than or equal to t_1 and less than t_2. For i=1, ts(O(i-1)) is assumed to be the current value of LC.

Now, let us define two macro operations on an object.

```
procedure revise(O, ts);
  begin
     O←(nil, O(0), O(1), ... , O(n(O)));
     n(O)←n(O)+1;
     for i=n(O) to 1 step -1 do
        L(O(i))←L(O(i-1));
     for i=n(O) to 2 step -1 do
        ts(O(i))←ts(O(i-1));
     L(O(0))←∅;
     ts(O(1))←ts;
  end;

procedure roll-back(O);
  begin
     O←(nil, O(1), ... , O(n(O)));
     L(O(0))←∅;
  end;
```

Lock operations are defined as follows. Associated with each read transaction T is a semi-open interval $[t_1, t_2)$ that is initially set to [0, +), where + denotes the positive infinity. This interval will be called temporal requirement, and be denoted by tr(T). Associated with each transaction T is a finite object set obj(O) that stores names of objects with a lock by this transaction. In the following definitions, T denotes the subject transaction.

```
procedure r-lock(O);
  begin
     find the minimum i≥1 s.t.
        tr(T)∩[ts(O(i)), ts(O(i-1)))≠∅;
     L(O(i))←L(O(i))∪{T};
     tr(T)←tr(T)∩[ts(O(i)), ts(O(i-1)));
     obj(T)←obj(T)∪{O};
  end;
```

The minimality of i is required to read the latest referable value of each object. The condition and the second last assignment are necessary to read out contemporary version values. Otherwise, a transaction may read values of different objects that did never exist at the same time. Such a problem will be referred to as a version consistency problem. Our protocol can ensure version consistency. This will be proved later in this section.

```
procedure w-lock(O);
  begin
     if L(O(0))≠∅
        then reject the request
        else
          begin
            L(O(0))←{T};
            obj(T)←obj(T)∪{O};
          end;
  end;
```

This procedure ensures mutual exclusion of simultaneous write lock requests.

```
procedure r-unlock(O);
  begin
     for i=1 to n(O) do
        L(O(i))←L(O(i))-{T};
     obj(T)←obj(T)-{O};
  end;

procedure w-unlock(O);
  begin
     roll-back(O);
     obj(T)←obj(T)-{O};
  end;

procedure commit
  begin
     if T is a read transaction
        then
          for each O in obj(T) do r-unlock(O)
        else
          begin
            ts←LC;
            LC←ts+1;
            for each O in obj(T) do
              if O(0)≠nil then revise(O, ts)
                          else roll-back(O);
          end;
     obj(T)←∅;
  end;

procedure roll-back;
  begin
     for each O in obj(T) do
        roll-back(O);
     obj(T)←∅;
  end;
```

The multiversion hierarchical lock protocol that uses the above lock procedures is described as follows.

Protocol

1. A write transaction can not refer to nor modify an object without setting a write lock on this object, while a read transaction can not refer to an object without setting a read lock on it.
2. Read transactions do not use w-lock nor w-unlock, while write transactions do not use r-lock nor r-unlock.
3. Each transaction locks an object no more than once.
4. Before locking an object, a write transaction must lock its parent object if it has one. A write transaction should not release a write lock on an object O before it has set all the necessary locks on the son objects of O. If it has locked all necessary son objects, and if it has never modified O and will not modify nor read O, then it may release the write lock on O by using w-unlock. Otherwise, it can release the lock after it has finished all update operations.
5. Each transaction must release all the locks it has set before its exit.
6. Each write transaction can issue no more than one commit command and any number of roll-back commands.

Any interleaved execution that follows this protocol is deadlock-free and serializable. These are proved as follows.

Theorem 4.2.1
Any interleaved execution that follows this protocol causes no deadlock.
proof For two write transactions T_1 and T_2, let $O:T_1 \rightarrow T_2$ denote that T_2 requested a write lock on an object O after T_1 had set a write lock on the same object. If there is a deadlock, there must be a chain such that $O_1:T_1 \rightarrow T_2$, $O_2:T_2 \rightarrow T_3$, ... , $O_n:T_n \rightarrow T_1$. If $O:T_1 \rightarrow T_2$ holds for an object O, then the fourth rule of our protocol implies that, for the parent object p(O) of O, $p(O):T_1 \rightarrow T_2$ also holds. Hence, for the root object O_r, $O_r:T_1 \rightarrow T_2$ must hold. If a deadlock occurs as the above chain, it must also hold that $O_r:T_1 \rightarrow T_2$, $O_r:T_2 \rightarrow T_3$, ... , $O_r:T_n \rightarrow T_1$. Since each transaction can set a lock on the same object no more than once, these relations lead to a contradiction. □

Theorem 4.2.2
In our protocol, a roll-back command can roll back any failed updates before the exit of the transaction.
proof Obvious from the definition. □

Theorem 4.2.3
Any interleaved execution of transactions following this protocol is serializable.
proof Let us consider interleaved execution of transactions T_1, T_2, ... , T_n. Let us define the transaction time tt(T) for each transaction as follows. If T is a write transaction, tt(T) is the LC value at the time when T issued a commit command. If T is a read transaction that finished with $[t_1, t_2)$ as its final temporal requirement, then tt(T) is defined as $t_1+e^{-id(T)}$, where id(T) is a positive integer that identifies T. This value is well defined since temporal requirements will never become empty. Let $T_{P(1)}, T_{P(2)}, \cdots, T_{P(n)}$ be a permutation of $T_1, T_2, \cdots, T_n$ satisfying that if $i<j$ then $tt(T_{P(i)})<tt(T_{P(j)})$. Then, the effect of the interleaved execution is equal to the serial execution $T_{P(1)}, T_{P(2)}, \cdots, T_{P(n)}$. This can be proved as follows. Let i be the maximum integer satisfying the following partial serializability condition: If $T_{P(i)}, T_{P(i+1)}, \cdots, T_{P(n)}$ had not actually modified any object, the execution would have produced the same effect as the serial execution $T_{P(1)}, T_{P(2)}, \cdots, T_{P(n)}$, except the values read out by $T_{P(i)}, T_{P(i+1)}, \cdots, T_{P(n)}$. Obviously, i must be greater than 1. If there exists no i less than or equal to n, then the theorem is proved. Assume that i is a positive integer less than or equal to n. The transaction $T_{P(i)}$ is either a read transaction or a write transaction.

Case 1 ($T_{P(i)}$ is a read transaction.) The read lock procedure r-lock ensures that every object version O(j) that $T_{P(i)}$ has read satisfies that

$$tt(T_{P(i)}) \in [ts(O(j)), ts(O(j-1))). \quad (4.2.1)$$

On the other hand, it holds for any $k<i$ that $\lfloor tt(T_{P(i)}) \rfloor \geq tt(T_{P(k)})$. Besides, the execution of $T_{P(1)}, T_{P(2)}, \cdots, T_{P(i-1)}$ can not produce any version whose time stamp is greater than $\lfloor tt(T_{P(i)}) \rfloor$. Therefore, the object version O(j) that satisfies (4.2.1) is the current version value immediately after the execution of $T_{P(1)}, T_{P(2)}, \cdots, T_{P(i-1)}$. Hence, the partial serializability condition also holds for i+1. This contradicts the maximality of i.

Case 2 ($T_{P(i)}$ is a write transaction.) Since $tt(T_{P(i)})>tt(T_{P(j)})$ holds for any $j<i$, it holds for any write transaction $T_{P(j)}$ satisfying $j<i$ that if both $T_{P(i)}$ and $T_{P(j)}$ set write locks on an object O then $O:T_{P(j)} \rightarrow T_{P(i)}$. This means that any modifications by $T_{P(i)}$ must have come after all modifications by write transactions in $T_{P(1)}, T_{P(2)}, \cdots, T_{P(i-1)}$. What remains to prove is that no read transactions in $T_{P(1)}, T_{P(2)}, \cdots, T_{P(i-1)}$ read version values modified by $T_{P(i)}$. Assume that $T_{P(j)}$ reads a version value O(k) that is produced by $T_{P(i)}$. Then it holds that $ts(O(k))=tt(T_{P(i)})$ and $tt(T_{P(j)})>ts(O(k))$, i.e., $tt(T_{P(j)})>tt(T_{P(i)})$. However, since i is greater than j, it holds that $tt(T_{P(j)})<tt(T_{P(i)})$, which contradicts the above relation. Therefore, the partial serializability condition also holds for i+1, which contradicts the maximality of i.

Therefore, there exists no such integer i, and hence the interleaved execution is totally serializable. □

The above discussion has assumed a simplified object model that allows monotonically increasing number of versions. Actually, an old version may be deleted if it is not subject to further references. For each read transaction T, let t(T) denote the LC value at the time when this transaction first set a read lock. Let tR denote the minimum t(T) value among any read transactions that are still being executed. Then, the execution of a commit command may delete all versions whose time stamp is less than tR, preserving only the latest of such versions for each object. It is obvious that these deletable versions are not subject to further references. This method keeps the maximum number of object versions within a reasonable range.

The multiversion hierarchical concurrency control is performed by Directory Searcher to decompose each transaction into correct concurrent execution of segment commands.

4.3 Directory Searcher

Interface/Supervisory Module and Query Processor require no innovative architectural technologies. They are necessary to provide a host computer with a high level interface to MPDC. Their software systems will not become much different from corresponding software modules of current relational database management systems. Query Processor transforms a given transaction into an optimized program that traverses tree structured segment directories and generates segment processing commands. The execution of this program is performed by Directory Searcher.

Suppose that we have two segment directories, as shown in Fig. 4.5, respectively for R(A, B, C) and S(D, E). These directories are based on colored binary trie schemes. For simplicity, each directory node is assumed to have only its current version. These example directories are much smaller than practical directories. For simplicity, attribute values are assumed to be 4 bit long. Some example query translations into segment processing commands are given below. They are performed by traversing the directories.

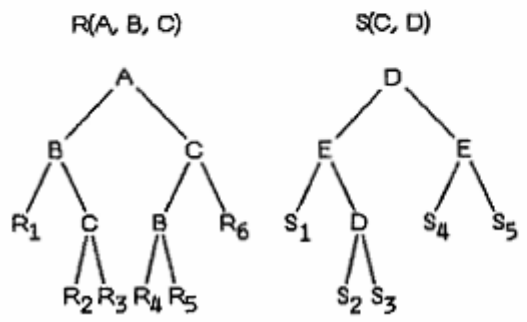


Fig. 4.5. Example directories in MPDC.

update Select tuples in R satisfying A=1000, B=0000, C=0000, and change their B values to 1000.

```
X1←[A=1000 and B=0000 and C=0000](R4);
X2←R4-X1;
◇X3←R5∪[B←1000]X1;
R4←put X2;
if ◇X3 consists of one page
 then R5←put ◇X3
 else
   begin
    segment-split(R5, attr, seg1, seg2);
    page-split(◇X3, attr, X4, X5);
    seg1←put X4;
    seg2←put X5;
   end;
```

The procedure segment-split(R_5, attr, seg_1, seg_2) will be executed by Directory Searcher during the execution of this program. Given an overflowing segment address, it returns the splitting attribute and two new segment addresses seg_1 and seg_2. The procedure page-split($◇X_3$, attr, X_4, X_5) will be executed by a segment processor. It will distribute tuples in $◇X_3$ into two single page variables X_4 and X_5, depending on the splitting attribute's values.

join

```
(R[B=0000])[A=D]S
X1←(R1[B=0000])[A=D]S1;
X2←(R1[B=0000])[A=D]S2;
X3←(R1[B=0000])[A=D]S3;
X4←(R4[B=0000])[A=D]S4;
X5←(R4[B=0000])[A=D]S5;
X6←(R6[B=0000])[A=D]S4;
X7←(R6[B=0000])[A=D]S5;
◇X8←condense(X1,X2,X3,X4,X5,X6,X7);
```

Actually, directory nodes have multiple versions. However, it does not much complicate directory search and generation of segment processing commands. The update example above, for example, changes the directory as shown in Fig. 4.6 if it causes an overflow. An underflow of a segment is

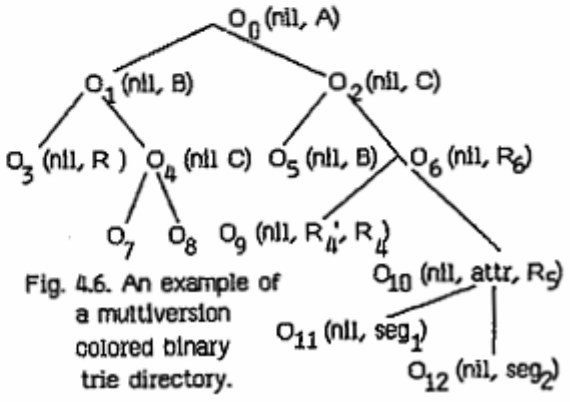


Fig. 4.6. An example of a multiversion colored binary trie directory.

recovered only by merging it with its brother segment if any. Therefore, some underflows are not recovered. However, our theory ensures that it does not cause serious problems. If an underflow occurs in O_7 and the merge of O_7 and O_8 can be stored in a single segment, then they will be merged into a new segment O_{78}, and O_4 will become (nil, O_{78}, c).

Directory Searcher is also in charge of concurrency control. Locks are set on objects in directories, following the protocol in Section 4.2. A read lock set on a segment is released when all commands that refer to this segment in the same transaction complete their execution and their completion tokens are all received from Data Subsystem by Macroparallel Data Flow Controller. A write lock on a segment is released in either of the following cases. If no command in the transaction actually changes this segment, it is released by a w-unlock command after all commands in the transaction that refer to this segment complete their execution. If a command changes this segment, it is released by a commit command after all updates and references in this transaction complete their execution. In a recovery routine of the transaction, a roll-back command is used to nullifies its failed execution.

4.4 Macroparallel Data Flow Controller

Macroparallel Data Flow Controller receives segment processing commands, dynamically constructs data flow programs of segment commands for segment transfers and page operations, sends active segment commands to Data Subsystem through the two commanders, receives completion tokens from Data Subsystem through the two watchers, and transfers activation tokens to next executable commands in data flow programs. To perform these operations, it has five tables, Transaction Table, Command Table, Segment Table, Page Variable Table, and Multipage Variable Table.

Segment Table describes, for each operand segment, a page variable assigned to it. When Macroparallel Data Flow Controller receives a segment processing command from Directory Searcher, it examines if the command has any operand segments that are not yet registered in Segment Table. If it has any, the controller assignes each of them a new page variable, and stores this assignment in Segment Table. Such a page variable is called a segment page variable. Operand segments in the original command are replaced with their corresponding page variables. A command thus obtained is called a page operation command. If the original command has such a new source operand segment, the controller generates a get command that requests transfer of this segment to the assigned page variable. If the original command has such a new destination command, the controller generates a put command that requests transfer of the assigned page variable to this segment. Put and get commands are called segment transfer commands.

The destination operand page variable of each command is allocated a free page in Shared Page Buffer when it is sent to Data Subsystem for its exexcution. Page Variable Table stores, for each operand page variable, its status, its allocated page address, a pointer to its reference list, and its reference count. The status field shows if the page value is already computed. Each page variable appears no more than once as a destination operand. When all operand segments in a command received from Directory Searcher are replaced with page variables, Macroparallel Data Flow Controller makes a new entry in Page Variable Table for the destination operand page variable. It also searches this table for its source operand page variables, which have been already registered in this table. It stores this command in Command Table with its subject transaction number and links this command to the reference lists associated with the source operand page variables' entries in Page Variable Table. The reference count of these entries are incremented by one. Their page allocation fields are set to nil.

Command Table is a waiting room for generated segment transfer commands and page operation commands. It stores a waiting command with its subject transaction number and as many token bits as this command's source operands. Associated with Command Table is the active command queue that stores addresses of the commands in this table that are ready for execution. When a command is stored in Command Table, the controller also examines its source operand page variables' statuses stored in Page Variable Table. If a page variable value is already computed, its corresponding token bit is set to one. Otherwise, it is set to zero. If all token bits of the command are set to one, this command is added at the end of the active command queue. Get commands are always added at the head of the queue immediately after they are generated. Each command becomes active when all of its source operand page variable values are already prepared in Shared Page Buffer, in other words, when all token bits of the command become one. When a command becomes active, it is added to the active command queue. Active commands are sent to Data Subsystem with their locations in Command Table. Before they are sent, their operand page variables are replaced with physical page addresses in Shared Page Buffer. Their source operand page variables must have been already assigned page addresses because of the single assignment rule and the activation mechanism.

Macroparallel Data Flow Controller replaces the source operand page variables with their corresponding page addresses by searching Page Variable Table. If a command has a page variable destination operand, the controller asks Page Buffer Manager to allocate one free page in Shared Page Buffer to this destination operand page variable. This allocation information is stored in Page Variable Table. If the destination operand is a multiple page variable, a list consisting of only this allocated page address is stored in Multipage Variable Table and a pointer that points to this list is stored in page Variable Table.

If a command is sent to Data Subsystem, it is removed from the active command queue. When Data Subsystem finishes execution of a segment command, it sends back a completion status code to Macroparallel Data Flow Controller either through Segment Transfer Watcher or through Page Operation Watcher. The completion status code includes the command address in Command Table. If the status code shows normal completion of execution, Macroparallel Data Flow Controller reads out the command address from the status code, gets its destination page variable from Command Table, searches Page Variable Table for this variable, and sets the corresponding token bits of all the commands that are chained from the reference list pointer field of this variable's entry in Page Variable Table. During this marking process, if the controller finds out a command whose all token bits become one, it adds this command to the active command queue. For each completed command, the controller also updates Page Variable Table entries associated with its source page variables. It decrements the reference count by one, and removes this command from the reference list. For each transaction, Transaction Table has a pointer that points to a chain of this transaction's variables, except segment page variables, in Page Variable Table. When a transaction finishes its execution, page allocation to the variables in the list linked from this transaction's entry in Transaction Table are all released, and these variables are deleted from page Variable Table.

When all pages in Shared Page Buffer are spent for page allocations, Page Buffer Manager can not allocate a new page to a new variable without releasing one page allocation. Its selection is based on the LRU algorithm. Suppose that the selected page is a segment page. If the transaction that requested the preparation of this page is already finished, the page allocation is released and this variable and the read out segment are deleted respectively from Page Variable Table and Segment Table. If the transaction that prepared this page is not finished or if the selected page is not a segment page, then this page value is saved into a work disk storage space provided by the disk subsystems. The destination address in the disk storage space is written in the page address field of Page Variable Table with a mark indicating disk storage space allocation. If an active command that is to be sent to Data Subsystem has a source page variable whose allocated page address is in the disk storage space, the controller asks Page Buffer Manager to allocate this variable a new page, sends a get command to Data Subsystem and put the object command at the end of the active command queue.

Macroparallel Data Flow Controller controls the activation of sufficiently large macro operations such as segment transfer or page processing. Therefore, the control overhead will be hiden by the concurrent execution of macro operations by Data Subsystem. An appropriate selection of segment size is required.

5 CONCLUSION

The massive parallel database computer architecture that has been proposed in this paper provides both fundamental technological breakthroughs in the performance enhancement and the capacity enlargement of database processing and a unified way of integrating these technologies into a database computer. Because of the 'Disk Paradox' pointed out by H. Boral and D.J. Dewitt (Boral and Dewitt 1983), it has been believed these days that the use of moving head disks as secondary memory devices puts it out of the question to design and to implement a massive parallel database machine.

The MPDC architecture has solved this problem as follows. It has decomposed database processing into two levels, i.e., directory search and segment processing. Every relation is divided into equal-sized segments. Corresponding to this two-level decomposition, MPDC consists of two subsystems, i.e., Data Subsystem and Control Subsystem. Data Subsystem is in charge of segment accesses and segment processing, while Control Subsystem is responsible to Data Subsystem for decomposing query transactions into concurrently executable segment processing commands. File segmentation requires file clustering schemes to increase file access locality and to decrease segment accesses and segment processing tasks. Control Subsystem uses adaptive multiattribute clustering methods called colored binary trie schemes. Data Subsystem consists of a pool of processors for segment processing, a set of disk subsystems, and a Shared Page Buffer shared by these modules. Segment processors use two types of bit-sliced VLSI modules for high speed processing of batch search and sort operations. These modules overlap their

processing with data transfer to and from them to make much use of page transfer time. Shared Page Buffer resolves memory access conflict problems in parallel processing and allows massively parallel processing based on macroparallelism among segment processing tasks. It allows 10^3~10^4 ports to concurrently access arbitrary pages without causing any conflict nor any access suspension. Some ports are connected to segment processors and others are used by disk subsystems. Shared Page Buffer does not only increase segment access speed but also decreases secondary memory access frequency. These two effects together with the provision of concurrently accessible multiple ports have solved the alleged 'Disk Paradox' of massive parallel database machines. Control Subsystem, on the other hand, uses a unified control algorithm that does not only manage adaptive segmentation but also efficiently and correctly control highly reliable interleaved execution of transactions. Massive parallel execution of segment tasks requires highly concurrent execution of segment read and segment write operations. Besides, high reliability requires a sound recovery mechanism that does not seriously lower system performance during its execution. Control Subsystem uses a multiversion concurrency control mechanism based on the colored binary trie schemes as a unified solution to segment management, concurrency control, and recovery. Activation of segment access commands and segment processing commands are controlled by a dataflow controller, which automatically controls disk subsystems to transfer segments to Shared Page Buffer prior to their processing. Each active command is sent to Data Subsystem and be executed either by a segment processor or by a disk subsystem. The use of auxiliary files for further speed up has not been described in this paper because such files are also considered as relations and can be treated by MPDC software systems. Such software systems as well as detail specifications of the data flow mechanism and total performance evaluation require further elaboration.

REFERENCE

Babb, E. Implementing a Relational Database by Means of Specialized Hardware. ACM TODS 4 1, 1-29, 1979.

Banerjee, J., Hsiao, D.K. and Baum, R.I. Concepts and Capabilities of a Database Computer. ACM TODS 3 4, 347-384, 1978.

Baudet, G. and Stevenson, D. Optimal Sorting Algorithms for Parallel Computers. IEEE Trans. Comput. C-27 1, 1978.

Bentley, J.L. and Kung, H.T. A Tree Machine for Searching Problems. Proc. Int'l Conf. on Parallel Processing, 257-266, 1979.

Boral, H. and Dewitt, D.J. Database Machines: An Idea Whose Time has Passed? Database Machines, Springer-Verlag, 166-187, 1983.

Chang, H. On Bubble Memories and Relational Data Base. Proc. 4th VLDB, 207-229, 1978.

Chang, J. and Fu, K. Extended k-d Tree Database Organization: A Dynamic Multiattribute Clustering Method. IEEE Trans. Software Engineering 7 3, 284-290, 1981.

Chen, T.C., Lum, V.W. and Tung, C. The Rebound Sorter: An Efficient Sort Engine for Large Files. Proc. 4th VLDB, 312-315, 1978.

Coulouris, G.F., Evans, J.M. and Mitchell, R.W. Towards Content-Addressing in Data Bases. Computer Journal 15 2, 95-98, 1972.

Dewitt, D.J. DIRECT-A Multiprocessor Organization for Supporting Relational Database Management Systems. IEEE Trans. Comput. C-28 6, 395-406, 1979.

Goke, L.R. and Lipovski, G.J. Banyan Networks for Partitioning Multiprocessor System. 1st Annual Symp. on Comput. Arch., 21-28, 1973.

Hirschberg, D.S. Fast Parallel Sorting Algorithms. CACM 21 8, 393-400, 1978.

Kung, H.T. The Structure of Parallel Algorithms. Advances in Computers 19, Academic Press, 65-112, 1980.

Kung, H.T. and Lehman, P.L. Systolic (VLSI) Array for Relational Database Operations. Proc. ACM-SIGMOD, 105-116, 1980.

Lawrie, D.H. Access Alignment of Data in an Array Processor. IEEE Trans. Comput. C-24, 1145-1155, 1975.

Muller, D.E. and Preparata, F.P. Bounds for Complexity of Networks for Sorting and Switching. JACM 22 2, 195-201, 1975.

Nassimi, D. and Sahni, S. Bitonic Sort on a Mesh Connected Parallel Computer. IEEE Trans. Comput. C-27 1, 2-7, 1979.

Oflazer, K. et al. RAP.3 - A Multi-Microprocessor Cell Architecture for the RAP Database Machine. Proc. High Level Language Computer Architecture, 108-119, 1980.

Orenstein, J.A. Multidimensional Tries Used for Associative Searching. Info. Proc. Lett. 14 4, 150-157, 1982.

Ozkarahan, E.A., Schuster, S.A. and Smith, K.C. RAP - An associative Processor for Data Base Management. Proc.AFIPS 44, 379-387, 1975

Parker, D.S.Jr. Notes on Shuffle/Exchange-Type Switching Networks. IEEE Trans. Comput. C-29, 213-222, 1980.

Pease, M.C. The Indirect Binary n-Cube Microprocessor Array. IEEE Trans. Comput. C-29, 213-222, 1980.

Preparata, F.P. New Parallel Sorting Schemes. IEEE Trans. Comput. C-27 7, 669-673, 1978.

Sagan, C. The Dragons of Eden. Random House, Inc., 1977.

Schuster, S.A. et al. RAP.2 - An Associative Processor for Databases and Its Applications. IEEE Trans. Comput. C-28 6, 446-458, 1979.

Stone, H.S. Parallel Processing with the Perfect Shuffle. IEEE Trans. Comput. C-20 2, 153-161, 1971.

Tanaka, Y., Nozaka, Y. and Masuyama, A. Pipeline Searching and Sorting Modules as Components of a Data Flow Database Computer. Proc. IFIP '80, 427-432, 1980.

Tanaka, Y. A Data Stream Database Computer. Japan Annual Reviews: Computer Science & Technologies 1982, OHM-North-Holland, 265-286, 1982.

Tanaka, Y. A Data-Stream Database Machine with Large Capacity. Advanced Database Machine Architecture, Prentice-Hall, Inc., 168-202, 1983.

Tanaka, Y. Adaptive Segmentation Schemes for Large Relational Database Machines. Springer-Verlag, 293-318, 1983.

Tanaka, Y. Bit-Sliced VLSI Algorithms for Search and Sort.Proc.10th VLDB,225-235,1984

Tanaka, Y. A Multiport Page-Memory Architecture and A Multiport Disk-Cache System. New Generation Computing 2 3, 1984.

Thompson, C.D. and Kung, H.T. Sorting on a Mesh Connected Parallel Computer. CACM, 20 4, 263-271, 1977.

Todd, S. Algorithms and Hardware for a Merge Sort Using Multiple Processors. IBM J. R&D, 22 5, 1978.

Uemura, S. et al. The Design and Implementation of a Magnetic-Bubble Database Machine. Proc. IFIP '80, 434-438, 1980.

Yasuura, H., Takagi, N. and Yajima, S. The Parallel Enumeration Sorting Scheme for VLSI. IEEE Trans. Comput. C-31 12, 1192-1201, 1982.