A PARALLEL EXECUTION MODEL FOR A DATABASE MACHINE WITH HIGH PERFORMANCES

Didier Donsez, Pascal Faudemay

Laboratoire MASI, Université Paris 6, 4 Place Jussieu, 75252 Paris Cedex 5, France
fax: 33 (1) 46 34 19 27

Abstract. *In this paper, we present a mixed MIMD / SIMD execution model for a reconfigurable computer. This model is adapted to the use of a specialized associative coprocessor, embedded in this host machine. A main characteristic of the model is that it uses four types of processes (decoding, calculus, coprocessor communication and transaction manager), and that in principle one process of each type is allowed on each processor. Time intervals are allocated to operations into partitions of the set of processors. Transfers are usually limited to identifiers, logical addresses and locks. Simulations display a high level of processors occupation. Therefore the machine yield may be very high, and the operations should be very fast.*

1. Introduction.

Databases seem to have very good prospects for the next ten years. Relational or object-oriented database systems [Codd 1970, Stonebraker 1976, Lécluse & Richard 1989, Abiteboul & Kanellakis 1989, etc..] offer a much greater flexibility than file systems. However, many applications still use simple files. To a great extent, this results from performance limitations of database systems, which are still rather far from real-time response times, specially in the case of large source relations. Numerous specific architectures, both hardware and software, have been recently developed in order to increase the performances of DBMS (data base management systems). These architectures rely on specialized circuits and highly parallel multiprocessors architectures.

The purpose of specialized processors or machines is to increase the performances by at least one order of magnitude, versus the general processors existing or currently used at the same time. Some specialized architectures, which we shall call co-architectures, are designed to be embedded in general host machines, and increase the performances of most general processors or machines, when used for database operations. In this paper we consider the use of an associative co-architecture called Rapid [Faudemay & al. 1987, Faudemay & al. 1988 a], embedded in a multiprocessor host machine.

A large variety of specialized circuits has been designed for database and knowledge base operations. Specific circuits have been designed for sorting [Lee & al. 1987, Kitsuregawa 1989, Tanaka , etc..], text retrieval [Haskin & Hollaar 1983, Takahashi 1987, Ben & Choi 1989, Lee & al. 1989, etc..] or more specialized operations such as aggregates calculus [Abdelguerfi 1989]. Some more general processors have also been proposed, such as Datacycle [Lee & Herman 1987], and to some extend Prolog processors. An other trend is the use of associative memories [Chisvin & Duckworth 1989], but most of them are still subject to important limitations such as a uniform record and field length within a relation, one byte words, etc... The use of specialized circuits in databases is sometimes criticized in the database community, based on the argument that the progress of general processors makes useless the design of specific circuits. However this argument can also be opposed to all research efforts done on ASIC circuits (Application Specific Integrated Circuits), which are developing very fast [Musgrave 1989, De Micheli 1988].

Multiprocessor architectures are either dedicated database machines, such as DBC 1024 (Teradata) , Tandem [Tandem 1988] or Copernique computers, or general host machines which may also be used for the embedding of a co-architecture. In these architectures, communications between processors are a major limiting factor of the power of the machine. Minimisation of interprocessor transfers has been the subject of numerous works based on hashing and clustering techniques in order to increase the locality of data [Fushimi & al. 1986, Cheiney 1986, Copeland 1988]. One target of the paper is to study a minimisation of the interprocessor communications, and also of the system calls. The increase of the memory size associated to each processor, and the use of non-volatile memories [Agrawal

CH2895-1/90/0000/0056$01.00 © 1990 IEEE

& al. 1989, Eich 1989], are also part of the solutions to reduce the needed communications bandwidth. We shall not consider them in this paper.

Rapid is an associative coprocessor with more general functionalities, and is designed to be used in a co-architecture within a general host machine. It is a parallel circuit, whose degree of parallelism only depends upon the technology and the number of components connected to the host machine bus. Rapid's instruction set includes relational algebra operations, plus text retrieval operations, aggregates and sorting. The object of this paper is the embedding of such a coprocessor in a highly parallel host machine. In RAPID-MP (Rapid embedded on a Multi-Processor Host), we embed one or several Rapid coprocessors in each cluster of the host machine. The operations take place where data and code are placed. Exchanges between processors are in principle limited to exchanges of identifiers [Abiteboul & Kanellakis 1989]. The performance objective of the architecture is to execute the operations in a duration which should remain comparable to that of the transfers of identifiers on the network.

In paragraphs 2 and 3 we present the principles of the Rapid coprocessor and of the target machines considered in this paper. These machines are limited here to the "reconfigurable" machines, based on a hierarchy of crossbars. In paragraph 4, we present the main choices of our execution model. Paragraph 5 presents the main lines of our simulation, and paragraph 6 its results. Paragraph 7 concludes.

2. Rapid principles

Rapid is an associative co-architecture for data and knowledge bases. This co-architecture is realized by the embedding of a fine grain parallel coprocessor, implemented in VLSI, within a host machine. The coprocessor is dedicated to parallel evaluation of logical formulas on records or tuples. Its data are tuples or more generally objects, and it returns the boolean value of a global formula, sequences of identifiers of the objects which satisfy such a formula, or the number of these objects [Faudemay & al. 1987].

Rapid coprocessor is mainly composed of an array of processing elements (PEs), in which the control and some data (the subexpression values) are propagated from one PE to the following ones, or for some problems, to the previous ones. PEs have a hardwired control, and a reduced instruction set which includes the relational algebra operations. It thus executes a very high level language [Faudemay & al. 1988 a]. A specific resolution structure which is dynamically reconfigured by data is distributed between PEs. It executes the query resolution by hardware, which is mandatory in this type of processor. It is therefore a database coprocessor, which cannot be used without a general processor and has a specific instruction set. However, the coprocessor instructions are written by the general processor into the coprocessor, and thus the coprocessor may be used with any type of host machine (it is a memory mapped coprocessor).

The feasability of the processor implied that it would be possible to integrate several processors per component, in order to obtain a satisfactory degree of parallelism. The first version of the PE occupies 3.5 * 3.5 mm2 in 2 microns CMOS technology with two layers of metal. Rapid has been designed in full custom VLSI, with lambda design rules. In this version, the cycle time of the PEs is presently 120 ns for the processing of a 22 bits word, including 16 data bits and 6 special bits (null bytes, etc..). The real duration of operations mainly depends, presently, upon the duration of the transfers on the host machine bus, and the host cycle time. As an example, the duration of a simple join of 1000 tuples by 1000 is presently about 22 milliseconds with a Motorola 68020 at 16 Mhz, according to a cycle by cycle simulation. Most of the time of the operation is due to the software layer operation. This layer is presently being implemented. In an ideal environment, the response time could be about 1 millisecond in a 1 micron technology. Further speed improvements are possible within the same technology.

The processed objects are transfered into the PEs without addressing them. The coprocessor is interfaced to the host as a memory, all input/ output registers of the PEs being at the same address. Each PE processes at most one predicate, corresponding to the comparison of an object element stored into the coprocessor, with an object element broadcasted on the host machine bus. If a predicate evaluation is distributed on several PEs, each of them evaluates one "long word" of each of both operands, the size of a "long word" being presently 32 bytes. Each long word is composed of a number of short words inferior or equal to the capacity of the local memory of the PE (16 words). It has been shown that this processor has the capacity of processing any data or knowledge base query.

The Rapid coprocessor is an extension of associative memories, but opposite to the classical associative memories, it makes possible the retrieval of tuples or records satisfying a complex logical formula. Other specialized database circuits exist but implement only part of its functions. The future evolutions of Rapid are directed towards a minimisation of the I/O transfers outside the component. A future target is an increase in the ratio between the number of PEs (or of useful memory) and the

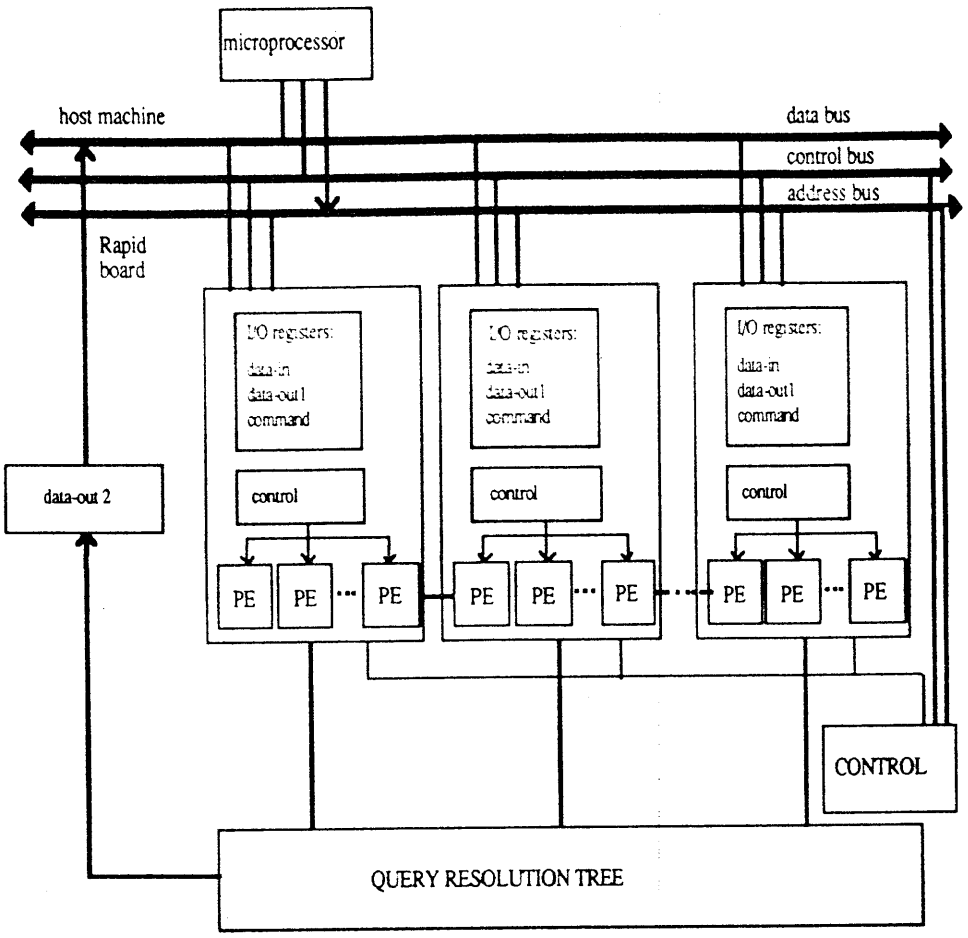


figure 1. Rapid coprocessor.

silicum area. In the long range, the purpose may be to keep part of the data within the circuit between operations. Presently, the data which are stored in Rapid are hashing buckets, and they only remain within Rapid during one step of an operation.

On monoprocessor host machines, the coprocessor operation is managed by a software layer, which interfaces it with the optimizer of a DBMS or KBMS, and with an adapted object manager. This layer is composed a single process. Its main functionalities are the distribution of the cpu time among transactions, an optional concurrency control method, the execution of multi-level hashing and of attribute calculus, and the transfers between the coprocessor and the general processor. Complementary functionalities may be added later, such as the management of hard real time [Stankovic 1988]. In order to obtain response times compatible with the coprocessor speed, this layer is implemented using a "factorization" method, where each logical step may be executed independtly from the others. A global presentation of Rapid methods and results is outside the scope of this paper.

Environment: reconfigurable machines.

In the Rapid project we are presently studying a Rapid-MP configuration, which results from the embedding of Rapid on a multiprocessor host machine. The multiprocessors which may be considererd belong to two categories: tightly coupled machines, where processor communicate through a common memory, and loosely coupled machines (sometimes called "shared nothing") where processors communicate by message passing. In this paper, we study the case of loosely coupled architectures.

These machines may also be divided into several classes, according to the way the messages are transfered between processors. One first technique is packet switching, where each packet is routed according to a specific address. This technique is frequently coupled with a multi-stages communication network, where messages are passed through several nodes of the network before reaching their destination. Circuit switching makes possible for several, arbitrary length messages to be transmitted

with only one routing. When this technique is coupled with a one stage network, the message can be transmitted in the minimum of delays, except the initial configuration time. In order to reduce the number of links and make possible any configuration of the network, an efficient way of achieving a one-stage circuit switching is through the use of a crossbar network, which is a mesh connected network with a controllable connection at each intersection. Such a network implements a reconfigurable architecture.

In the reconfigurable machines which we study, any topology of the network may be implemented. Though the change of the state of one intersection of the crossbar may be very fast, the partial or global reconfiguration usually implies some non-conflicting and possibly fault-tolerant routing. Therefore the duration of a reconfiguration is not neglictible, and may take one millisecond or more. Reconfigurations imply the use of synchronisation points, which may occur at some steps of a parallel algorithm (semi-dynamic reconfiguration). The use of single stage, circuit switching circuits, combined with the hardware facilities for message passing of some processors such as the transputers [Inmos 1988], may offer important progress possibilities. However dedicated parallel methods should be designed to make the best use of these architectures. This type of machine is represented by the SuperNode machine designed during the Esprit project P1085 [Harp 1987, Whitby-Strevens 1988], and by the SuperCluster from Parsytec [Kübler 1988]. The SuperNode machine is also characterized by a control bus for synchronization purposes.

The processors used in these machines are transputers with high throughput links making possible a fast message passing (presently 4 links with a bidirectional throughput of 24 Mbits / sec on each link). Due to the limited size of present crossbars, several crossbar levels are needed in order to connect a large number of these processors (up to one thousand). On these machines, a set of processors connected by a single level of crossbar is called a cluster. Our study has been specially directed towards the SuperNode machine. In this machine, a cluster (called T-Node), groups 16 transputers connected by a first level crossbar. This crossbar is made of two switches, each having 72 * 72 links.The crossbar is controlled by two more transputers, one of which is also the master of the control bus connecting all transputers of the T-Node. The backplane of a T-Node offers 7 slots, 2 of them being usable for two Rapid boards, without any change in the organization of the host machine.

Two main difficulties of the use of SuperNode for high performance database operations result from the duration of the reconfigurations, and from the multilevel organization of the machine. In this machine, the frequency of reconfigurations has a similar impact on the available communication bandwidth as the number of messages in other architectures. Though the use of relatively frequent and semi-dynamic reconfigurations increases the interest of the architecture, the duration of the reconfiguration steps must be limited in order to make the maximum use of the communication throughput. This limitation may partly be reduced by changes in the reconfiguration methods, but also by an improvement of the methods used by the applications or the system. Another limitation results from the multilevel organization of the crossbar network, as the average througput between two processors belonging to different clusters is smaller than the throughput inside a cluster. This characteristic is common with other hierarchical architectures, based e.g. on busses, but is more limiting when the number of processors may be very large. These two considerations have motivated our choice of a specific execution model for an optimized utilization of Rapid on the SuperNode.

The parallel execution model.

Principles

Our model executes transactions in parallel. A transaction is composed of a sequence of retrieval and update queries, each of which being partially ordered within a tree. Some operations of a transaction may therefore be executed simultaneously (intra-transaction parallelism). An operation is activated when its source relations already exist (permanent relations), or have been completely produced by another operation or a step of a recursive operation (temporary relations). Such an operation will be said to be "ready" to execution. This MIMD (Multiple Instruction Multiple Data) approach is a dataflow one [Boral & Dewitt 1980, Gajski & al. 1982].

These operations are generally done on the processors in whose local memory the data are stored. We try to limit as much as possible the interprocessors transfers. Using multiple copies of relations placed on independent criteria as in Bubba is one way for this [Copeland 1988]. Another copy of each relation or more may also be placed on a complementary criterium to ensure the availability of the relation in case of a breakdown of one node [Cheiney & al. 1986]. An operation is divided into local operations which execute in parallel, each on one of the processors where the relevant copy of the source relations are distributed. Each local operation is also divided into

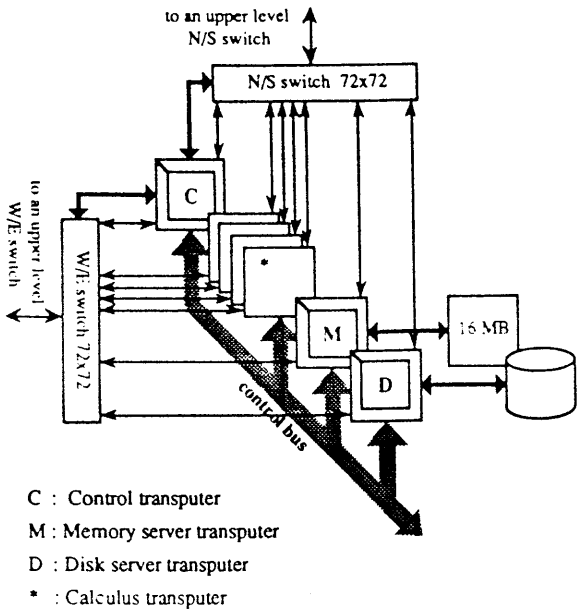


figure 2. T-Node Organization

sub-operations in order to manage the pseudo-parallelism between transactions, without commuting processes (multithreading function of the Rapid software layer). We assume that when a join operation is done on two relations, there is at least one copy of each source relation which is placed on a common set of processors according a hashing function of the join attribute. In some cases this condition is not satisfied and data transfers must occur. The execution of an operation is thus executed in an SIMD (Single Instruction Multiple Data) mode, each local operation being done on a different processor. Due to the specificities of Rapid and of the interconnection network, we avoid to execute two local operations in parallel on the same processor in different processes.

Data transfer on the interconnection network is done most of the time between two groups of processors. The cost of these transfers must include that of the communication itself, which is linear to the number of tuples, and an overhead corresponding to the reconfigurations. This overhead is independent from the number of transmitted tuples, but grows with the number of communicating processors (see example in figure 3). Therefore, depending on its size, a relation may have to be placed on a fraction of the total number of processors of a cluster, in order to limit the reconfiguration duration. We shall use the name "partition" for the group of processors where a given copy of a relation is distributed. The size of a partition is chosen at placement time. It depends upon the average characteristics of the operations which use it. This size will be

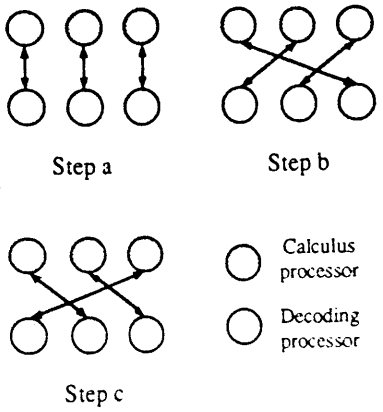


figure 3. Reconfigurations as a function of the number of processors (example with one link per processor)

discussed later. As our partitions have variable sizes, it is necessary to use the processors which do not belong to the partition of a current operation, for others simultaneous operations. This is a main justification for the utilisation of a mixed MIMD / SIMD execution model, which would not be as necessary if all relations would be partitioned on a cluster.

Logical addresses and concurrency control management.

Our execution model directly supports the notion of an object identifier [Abiteboul & Kanellakis 1989], as does Rapid's hardware. Each object is localized by a logical identifier, sometimes called "surrogate", which is a special kind of logical address which does not vary when the object value is modified. An identifier may be implemented as a logical address of a second level logical address, which is only invariant between two modifications of the object. When an object value is modified, its page number may be changed in the second level logical address (due to space or placement considerations), but this second level logical address may me modified in place. Therefore the identifier never changes. Another aspect of identifiers is that they use logical addresses adapted to databases, which have a structure "<page number><object number>", and not "<page number><offset>" as in classical virtual memory management. This results from the necessity of compacting the pages without changing the logical addresses. A consequence of this structure is that classical MMUs (memory management units) cannot be used with the same efficiency in this environment as in others, however we shall not deal with this problem in the paper.

In Rapid software machine the logical identifier can be

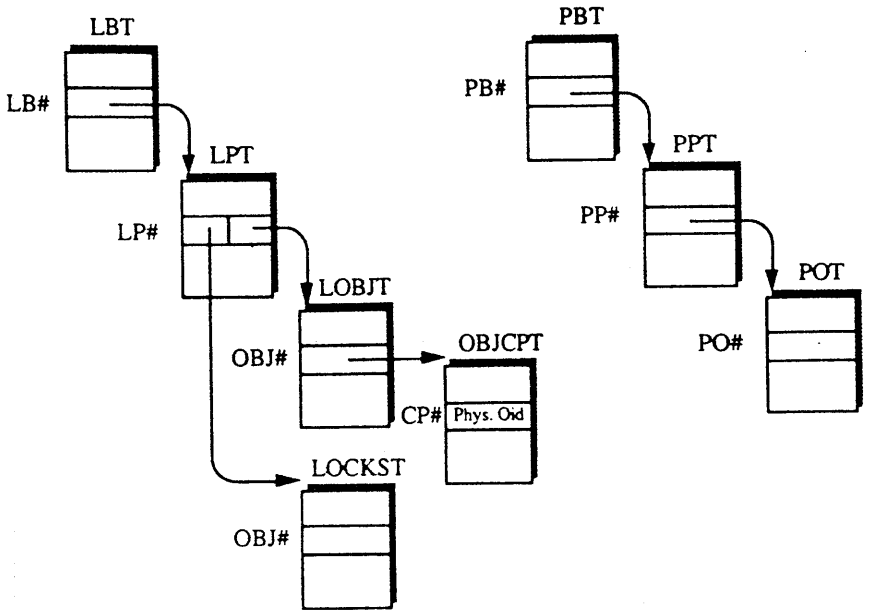


figure 4. system tables for identifiers translation.

LBT: Logical Bucket Table
LPT: Logical Page Table
LOBJT: Logical Object Table
OBJCPT: Object Copies Table
LOCKST: Locks Table
PBT: Physical Buckets Table
PPT: Physical Page Table
POT: Physical Objects Table

decomposed into a segment number, a logical bucket number, a logical page number, a logical object number and a copy number (and in some cases, a version number). A lock may be associated at the object level to each logical object number. The result of the logical identifier translation is a physical identifier. The decoding is done through a hierarchy of tables [Klinkhamer 1969]. The physical identifier is a classical database logical address, equivalent to the <page number><object number> structure of Ingres [Stonebraker & al. 1976]. In the case of Rapid it is composed of a segment number, a physical bucket number, a physical page number, a physical object number. The decoding of both logical and physical identifiers is represented in figure 4. Logical and physical bucket numbers are used to limit the size of each table and to allow the placement of tables and data.

Logical tables and locks tables are placed on specific processors according to the identifier value, while physical tables are placed according to the object value on the same processors as the corresponding data. There is usually no possibility of placing all logical tables on the same processors as the corresponding data, as logical identifiers are independent of the object values, which are used for the placement of data. Logical tables are placed on the value of the identifier, and not on the value of the corresponding data. Therefore they are placed into specific "decoding" partitions, while data are placed on "data partitions". A main issue in the model is to limit the number of messages between data and decoding partitions. A complementary aspect is that decoding partitions use much less memory than calculus partitions. In order to have a good occupation ratio of memory, we are therefore led to place calculus and decoding partitions on the same processors, but to synchronize the use of a given calculus partition with that of the corresponding decoding partition.

The proposed concurrency control method is the method of "counter-locks" [Faudemay & al., 1988 b]. This method is related to the two-phase locking (2PL), but each object lock makes possible the management of a nearly arbitrary number of

simultaneous transactions. The lock is accessed during the process of decoding of the logical identifier, and therefore the setting and unsetting of a lock is much faster than with the classical hash-based methods. Therefore it becomes possible to use object level locks even during set oriented queries (such as relational queries), and not only for debit-credit transactions. Using an object level locking method is very useful with main memory oriented databases (for a survey, see e.g. [Eich 1989]), where relevant data are often spread on many pages.

Use of several processes per processor or not

In a general solution, operations are started as soon as their data are ready. The operation is divided into identical processes which execute the operation code on a different processor of the partition. Each processor is then loaded with many processes corresponding to various operations and transactions. The time-sharing between processes is done by the operating system, which is frequently called. It has been shown that the workload of the operating system has a great impact on the performances of a multiprocessor system.

In order to reduce the load of the operating system, we have much simplified the placement and allocation of processes on the machine. In the normal mode of operation (in absence of a processor failure) only one process of each type is executed on each processor, except possibly from communication processes. The distribution of the workload between transactions is implemented within each process, by a multithreading capacity which is added to Rapid software functionalities. Operations are splitted into atomic sub-operations, and each process sequentially executes sub-operations from varied transactions. The allocation of sub-operations to processes takes into account fairness considerations. The problem is then to realize an efficient and fair allocation of sub-operations. An operation or a sub-operation is started only if all the processors of its partition are free. A priority mechanism guarantees that each sub-operation is executed in finite time, and generally within controllable duration limits, as the model is intended to allow real-time processing in the future. The atomic execution of each sub-operation guarantees that no loading of the coprocessor will be cancelled by the loading of another operation, which could produce performance problems. Our solution is more a controlled approach of the MIMD model than a chaotic one.

Process Placement.

In our model, processes belong to one of 4 types: the transaction management processes (TM) start the operations and suboperations on the corresponding processors, the calculus processes (CP) execute the operations, the decoding processes (DP) translate logical into physical identifiers and check and update the locks, the coprocessor communication processes (CC) exchange messages between the coprocessor and the calculus processors. Executing in parallel one process of each type on each processor would be possible, but does not seem desirable. If a message is sent to a processor which does not currently executes the corresponding process, the sender will wait or a context switching will have to occur. We have chosen to execute one type of process on each processor during relatively long time slices controlled by the TM. One processor per cluster is dedicated to the transaction management (TM). Other processors are dedicated to decoding partitions or to the operations. The coprocessor communication processes (CP) are placed on the transputers which manage the Rapid boards. These boards are placed on the two free slots of the T-Node. In order to maximize the workload of each processor, the processors dedicated to the calculus may be more or less numerous than those dedicated to the decoding. In order to make the transaction management reliable, the TM process may be transfered to another processor in case of a failure of its normal processor. Auxiliary processes, which we have not dealt with here, manage the transactions at a multi-cluster level or manage the communication with groups of users.

Partitions sizes of relations.

The power of the Rapid coprocessor makes possible the management of tuples at approximately the same speed as the tuples transfer. When considering the duration of each of these operations, the volume of data transfered for a 8000 * 8000 tuples join on a 32 bits attribute may be estimated as follows:

* oids decoding: 16000 words
* transfers to the coprocessor: 8000 words
* transfers from the coprocessor: 16000 words

which represents about 160 k bytes. If each processor is connected to another one through its 4 links, its maximum communication throughput is 96 Mbits / sec (96 Mbytes / sec for 8 calculus processors) and the duration of the transfer is less than 2 milliseconds. The minimum duration of the operation on Rapid boards is presently 1 millisecond for 2000

source tuples on one board, and is linear vs the number of tuples. These evaluations do not include the calculus time spent by the transputers for the communications and for the operation preparation, which may change these figures. If we consider the Mips ratio between a transputer and a 68020, the operation preparation should take about 3 milliseconds on each transputer. We have not yet estimated the cpu time spent by the transputers during the communications, which is not neglictible. However the Mips ratio only gives a raw indication, and a precise evaluation of these parameters will need the implementation of a more accurate evaluation model, which we intend to build in a second step.

Therefore the duration of each step of the above operation (8000 * 8000 tuples join returning 8000 tuples) is as follows:

* operation preparation	3 ms / calculus processor
* communications	2 ms / calculus processor
* total duration	5 ms

in parallel with the two previous steps:

* operation duration on each Rapid board: 4 ms / board

therefore the steps executed by Rapid can be done in parallel with the calculation and communications steps, and the average duration of a join on a T-Node should be about 0.3 microseconds per source tuple. This is rather good if we consider that the operations are done in a virtual memory environment, though most data are assumed to remain in memory, and that the methods are compatible with object levels locking.

In this evaluation, we consider that communications always occur between the same pairs of processors, which is not the case. The communication part of an operation is composed of a sequence of transfers, separated by links reconfigurations. A main issue is the minimization of the ratio of reconfigurations in the total duration of operations. This ratio mainly depends upon the distribution of relations among processors, and upon the number of calculus and decoding processors which communicate during an operation. A critical parameter is the size of the calculus partition, for a given size of a source relation,

This partition size is determined by an estimation of the size of the source relations of the operation, according to the permanent relations sizes and the average selectivity of the previous operations. If a relation is placed on a partition in order to optimize some join, its distribution is chosen as a function of the average number of source tuples of this join, weighted by the frequency of the operation instances. The validity of using this average number is one of the issues of the simulation, and other distribution characteristics might also be found useful.

As there are in most cases several partitions of each size , we must then chose a partition for the given relation copy. A first criterium is that there must be a copy of the relation on each partition where there is another relation, which possesses some foreign key belonging to the considered relation. Another criterium is to reach an equilibrium of the workload of all processors, during the execution of average sets of queries. Therefore the distribution of relations among partitions may be decided as a function of the frequency of access to each relation, or the average frequency per byte of memory [Copeland 1988]. The combination of these criteria with the choice of partition sizes within a limited set of partition sizes has not yet been fully studied. As a whole, the problem of the mutual placement of relations in the target machine will be the subject of further studies.

Another aspect of the problem is the distribution of partitions among processors. The constraints of our execution model imply that a processor may not be occupied by two operations in the same time. If two sub-operations executed in the same time have one processor in common, one of these operations will delay the other, due to the SIMD aspects of the operation execution. This phenomenon will increase the ratio of time during which processors are idle, and decrease the global occupation rate of the processors. In order to limit the ressource conflicts between operations, relations are placed on partitions which do not share processors. We define a hierarchical organization of the partitions, where each partition may be included in only one partition of the immediate superior size, and partitions of the same size have no processor in common.

Reservation method.

Our model uses a mixed approach, which combines medium grain SIMD and coarse grain MIMD. Several operations may be executed simultaneously by the machine on disjoint partitions of the processor. An operation is activated only if all processors of its partition are free. We want to define a mechanism for the allocation of operations to free partitions, during the time needed by the operation. The reservation may be done immediately or for a future time interval. This mechanism relies on the fact that the duration of operations using Rapid is efficiently predictible. This property may also be used in the future to define a real-time parallel database system using Rapid [Stankovic 1988]. The idea of allocating time intervals on partitions to operations makes the reservation method very

comparable to memory allocation problems, though some complementary mechanisms must guarantee that the partition is effectively free before the operation starts.

Several criteria must be satisfied by the reservation method:

(a) minimal fragmentation. We want to limit the time fragmentation of free processors in order to allow large operations to be allocated to these processors in finite time. The smaller the partition of an operation will be, the easier this operation will have the possibility to access it and to fragment the free time intervals of the larger partitions which contain it.

(b) fairness of access of different transactions: no transaction should reserve a partition for more than a certain amount of time, or benefit from more than a certain occupation ratio of this partition considered for a sufficient time duration, except if it has some further defined priority level.

(c) waiting operations must be efficiently restarted when a free partition becomes available.

In order to satisfy these three criteria, we have defined an allocation method of processors which is based on an anticipated reservation of partitions. In a first step, we present a general allocation method which is independent from the data structures used to store the operations. Then we shall define a more efficient method. In the general method, the reservation queries generated by the ready operations are stored in waiting queues associated to each active transaction, in order to be able to give the priority to each transaction using some round-robin method. For each reservation query, we try to find in the future a period of duration dt during which the processors of the needed partition part are free. We use two levels of priority. One corresponds to an imperative priority, for an operation which is allocated in the first available interval, whatever the situation of the other operations. The other level is simple priority, which determines the order in which the operations are allocated if some other conditions are satisfied. Each level of priority is represented by a token, which is held by one of the transactions. The types of priority which we use may also be helpful in the future to manage real-time database systems [Lee 1989], though this is not the case in the present version of the model.

The allocation is done with an imperative priority for the first operation of the transaction which has the imperative token. Presently, this first operation may be defined by classical query optimizers. For this operation, we allocate the first relevant time interval of the corresponding partition. The end of this time interval becomes the "limit-date" of what we shall call the "near" future for this partition (and in the present version, for the whole cluster). If this limit is too near, a further limit may be defined, according to a "minimum distance of the near future" (figure 6).

We then try to make the best possible occupation of the processors until this time limit. We try to allocate other operations (belonging to the same or the other transactions). In order to limit the cost of this allocation and satisfy the priority conditions, the availability period for these operations is seeked only in a "near" future, which is at most the "limit-date". The allocation query is satisfied if the partition is free during this period. If it is not the case, the allocation query is kept waiting again. In order to satisfy the priority conditions and the possible operations deadlines, we first try to allocate the first operation of each transaction, then we try to allocate the following operations, by passing the second level priority token.

When the "limit-date" of the near future becomes near enough from present time, we start a new imperative allocation of the first operation of the transaction having the first priority token. In a future version, imperative allocations could also be started by deadline considerations, thus decreasing the occupation ratio of the processors in order to satisfy real-time constraints. After this imperative allocation, the second level priority token can be either reset and given to the next following transaction, or kept in its current place.

The time reservation parameters are then:

* ddec, the interval before the limit-date which starts the exploration of the further future
* dlim, which represents the minimum duration of the near future.

A simplified version of the algorithm is given in figure 5.

This general solution has an inconvenient which is a non neglictible cost of the seek for free partitions in the future. In many cases, the waiting queues of allocation queries may not be all examined during the time explorations. It is therefore desirable to found a solution which makes possible to examine all the waiting allocation queries. For this reason, we propose another reservation method. In this version, we no longer seek for free partitions for waiting operations, but we directly seek for waiting operations which can occupy free partitions. In order to have a direct access to the description of the waiting operations, the operations of a partition are stored according to their partition number and their duration. Several classes of duration are defined, and the retrieval mechanism is much similar to that of the retrieval of holes in the process of memory allocation.

```
while TRUE
 do begin
    /* reservation of the partition having the priority */
        priority_date := seek( priority_op , interval [present_date,infinite[ )
        reservation( priority_op , priority_date )

    /* new "distant" future and new "near" future */
        limit_date := max( present_date+∂limit , priority_date+priority_op.duration )
        starting_date    := limit_date - ∂starting_delay

    /* we try to reserve the new near future for other waiting operations */
        priority_op := current_op := next_op( priority_op )
        while present_date < starting_date
         do begin
        /* We look for the partition corresponding to the current operation */
            current_op_start_date :=
                seek( current_op, interval [ present_date,limit_date ] )
            if current_op_start_date ≠ 0
                then    reservation( current_op , current_op_start_date )
            endif

        /* we begin the next operation */
            if current_op_start_date ≠ 0 and priority_op == current_op
                then    priority_op     := current_op := next_op( current_op )
                else    current_op      := next_op( current_op )
            endif
          end
 end
```

---

seek(Op , interval [starting_date , ending_date])
retrieves a period equal to that asked for by Op

reservation(op , date)
reserves the partition asked by Op

next_op (Op)
returns the operation following Op in the relevant priority order.

---

figure 5. reservation method.

We use this mechanism for the allocation of operations having the second level priority token. The mechanism for imperative allocation is nearly unchanged. When a partition is free for some duration, we first examine if an imperative allocation is nearly needed. In that case we first examine if the corresponding operation may fit in this time interval. Otherwise we successively look for a possible operation to execute in this partition, in each duration class starting from the nearest smaller class. In this class we allocate the first operation of the waiting queue, which corresponds to the operation having the higher priority (usually, the older operation). In general, this allocation does not occupy the whole time interval, and we then try to occupy the rest of the time interval with operations needing smaller intervals, or smaller partitions included in the considered partition. The priority mechanisms seem therefore different from those of the general version, and we intend to examine in further papers the effect of these two mechanisms on the fairness and possible time constraints of the operations. In this version, the reservation cost, which is a parameter of the simulation presented in the following paragraph, may in general be neglected.

Simulation method.

With this simulation, we try to know what is the processors occupation ratio which may be obtained by the time reservation method. Our purpose is to maximize the reservation and occupation ratio. The reservation ratio is the ratio between the total reserved time of the processors and the total duration of the experiment on all processors. The occupation ratio is the ratio of the time really used by operations (excluding the reconfiguration time), vs the total cpu time. As the fairness of the method seems guaranteed by the algorithm, we do not study it in this first series of experiments. Therefore it is not necessary to

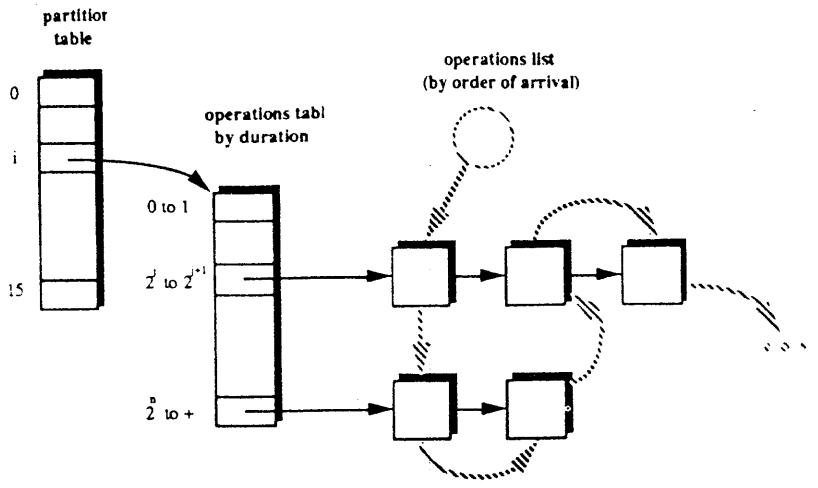


figure 6. Data structures for the direct placement of operations

consider the workload at the transaction level. We only consider series of operations placed on several partitions, without mapping them on transactions.

The main characteristics of an operation are its duration, its partition, and its arrival time. The duration of an operation is calculated using the size of the corresponding permanent relations, and the selectivity of the previous selection operations. We limit ourselves to consider queries of the "selections-join" type, or equivalent more complex queries. In order to have a simple enough simulation, we consider only two distribution laws, the distribution law of the sizes of permanent relations and the distribution law of the total selectivity of the previous operations. These two laws must respect the hypotheses on the data placement, and the equilibrium of the processors workload. As the duration of an operation is considered to be linear in the number of source tuples (which is approximately true with Rapid), we only consider one source relation per operation, and only one type of operation, which corresponds e.g. to a join. The operation calculus is considered to be composed of a decoding step followed by a calculus step. The transfer operations, which are supposed to be done in parallel, are neglected and will be the subject of further simulations.

For each operation the partition size has to minimize the reconfiguration ratio. The best partition size for a copy of a permanent relation is then a function of the size of the source relations of the operations using this relation. The distribution of the permanent relations is chosen according to four size classes of permanent relations, each one corresponding to a partition size in case of a selectivity equal to 1. The simulations have been done with various values of this distribution, which will be displayed in the next paragraph. The selectivity of the previous operations is defined by the average value of this selectivity. Three classes of selectivity have been defined, corresponding to average selectivities 0.5, 0.05 and 0.005. In our present simulation, each of this classes has the same probability (1/3). In the simulation, each operation has a previous operations selectivity, which is taken in one of these 3 classes with probability 1/3, and varies around the class average s* with a uniform distribution in the interval [s*.(1-a), s*.(1+a)]. When chosing a = 1, the distribution is uniform in the interval (0, 200%). Therefore each class has a non-empty intersection with the others. The existence of a large range of operation sizes in each partition is needed for the significance of the simulation. The partition number is chosen randomly in each partition class, characterized by its number of processors.

The operations are generated with a creation time which follows an uniform law, with a given creation rate. The creation rate is tuned in such a way that there is no long term increase of the waiting queues, i.e. the creation rate is just under the saturation level of the machine. It thus seems useless to consider complex waiting queues modelisations , and more complex creation laws, such as Poisson laws. The size of the waiting queue is given for various simulations in the next paragraph, and shows that the parameters do not lead to a saturation of the machine (figure 7).

The size of the operation partition, and that of the source relation, enable us to calculate the duration of the operation, including the reconfigurations duration. Due to the variability of

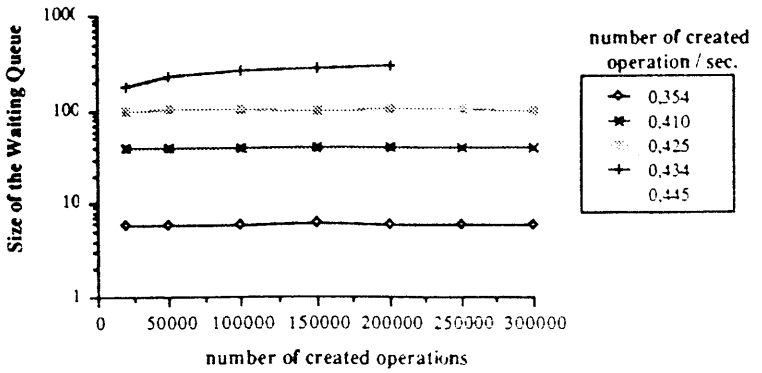


Figure 7. Size of the waiting queue according to the operation throughput (simulation type 3, I4 = 0.25)

the selectivities, some operations would have a reconfiguration ratio which is above 20%, which we have chosen as the desirable limit of the reconfiguration ratio. In this case we group the operation with other ones, which use the same configuration but not the same source relations, in order to avoid concurrency control problems. We have not yet evaluated if operations groupings may be limited in some cases by this condition. If we would have to drop the groupings in some cases, it is clear that the occupation ratios would not be as good as in our first results.

Simulation results.

The simulation results are presented as a function of the size distribution of the permanent relations, which belong to four classes: small relations (I1), medium size relations (I2), large relations (I4), and very large relations (I8). We do not intend to make assumptions about which relations sizes are probable in real applications, but we only distribute relations sizes according to the corresponding partition size. With an average operation selectivity of 1, relations of class I1 are distributed on only one processor, while relations of class I8 are distributed on 8 processors. We assume that 8 processors is the maximum size of a calculus partition in the present organization of the machine. Relations sizes in each class could be much increased if we assumed a higher selectivity of current operations (and specially selections), which means a selectivity much smaller than 1. Simulations are divided into 3 groups. In the first one (type 1), the total number of tuples of small relations is 0.375 of the total, in the second one (type 2), it is 0.200, in the third one (type 3) it is 0.100. The total ratio of the number of relations belonging to the very large relations class (I8), is always 0.011. The ratio of the I4 class vs the total number of relations is S. The ratio of the class I2 corresponds to the remaining tuples. In each group of simulations we have presented the results as a function of S.

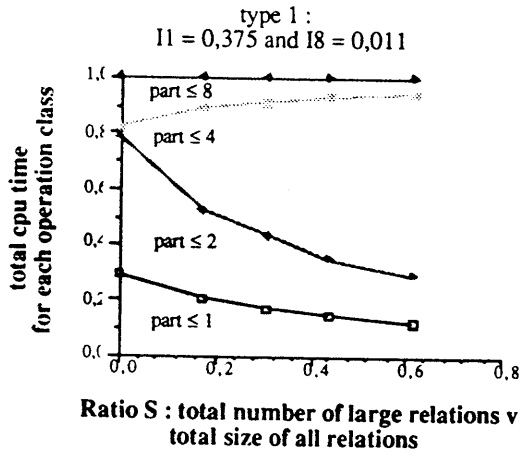
The first series of figures (figure 8) displays the total cpu time for each operation class. Each operation class is characterized by the number of processors of their partition. As an example, part 4 operations use 4 processors. The cpu time is cumulated for a given class and the smaller ones, therefore returning the cumulated value for class ≤ i, where i is the number of a class. The total cpu time for a single class is thus the distance between two of the curves. In type 1 simulations, the ratio of part ≤ 8 grows naturally as this part increases. In type 2 and 3, we can observe another phenomenon. The ratio of operations belonging to class 4 first decreases, then increases again. This results from the fact that the partition size of an operation does not only depends from the size of the corresponding permanent relations, but also from the selectivity of previous operations. Part of the operations using data derived of I8 type of permanent relations are executed in partitions of 2 or less processors. Therefore the ratio of the operations done on these partitions increases in a first time. An analytical presentation of this phenomenon should be given later.

The next figures (figure 9) display the operation throughput at saturation level and the size of the waiting queues according to the same parameters. As the ratio of large relations increases, the average size of operations also increases and the saturation throughput decreases in terms of numbers of operations. This result is rather straightforward. In the same conditions, the size of the waiting queue tends to increase. This seems to be due to the fact that with larger operations, the placement of small operations tends to become more difficult, and a larger waiting queue is necessary in order to obtain a good placement of these operations. The waiting queue increases as long as a significant ratio of the operations cannot be placed fast.

The last series of figures (figure 10) displays the

evolution of the reservation ratio and occupation ratio when the ratio of the large relations increases. Though the placement of the small operations becomes more difficult until the waiting queue is large enough, large operations make a good use of the largest partitions, which returns a higher reservation and occupation ratio. This seems to demonstrate that our reservation method enables large operations to be placed with a good respect of their priorities, thus guaranteeing the efficiency of the placement. The occupation ratio, which is the main result when considering the extimated yield of the machine, is always larger than 0.76, and may reach more than 0.82. This results from the good reservation ratios, and from the fact that we group operations when the reconfiguration ratio grows above 20%. Possible difficulties for this grouping may arise from concurrency problems, and this issue will be addressed in further studies. As a whole, the simulation results seem to indicate that the proposed execution model is worth being studied.

figure 8. total cpu time per operation



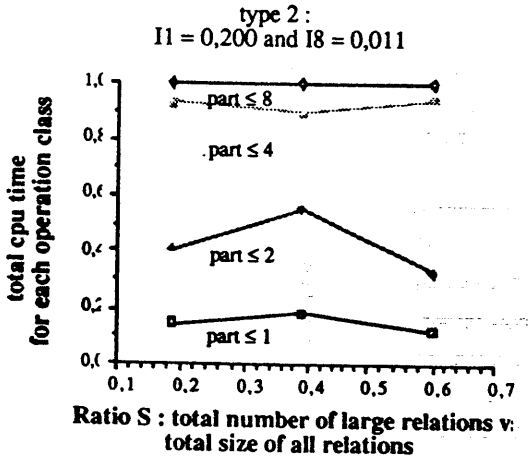


figure 8 (continued). total cpu time per operation

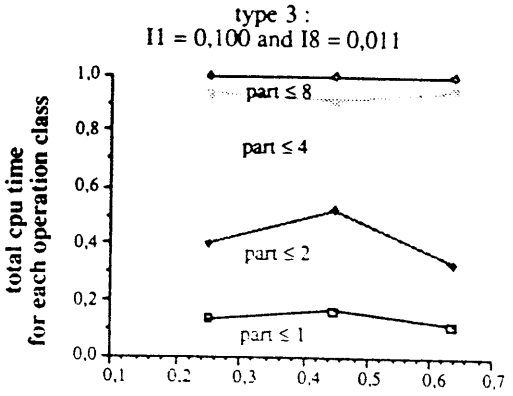


figure 9-a. saturation throughput

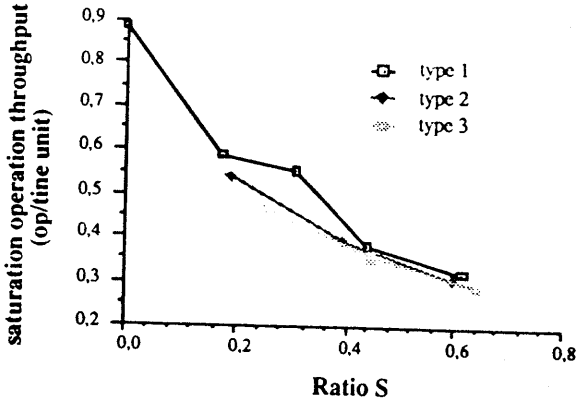


figure 9-b. waiting queues length

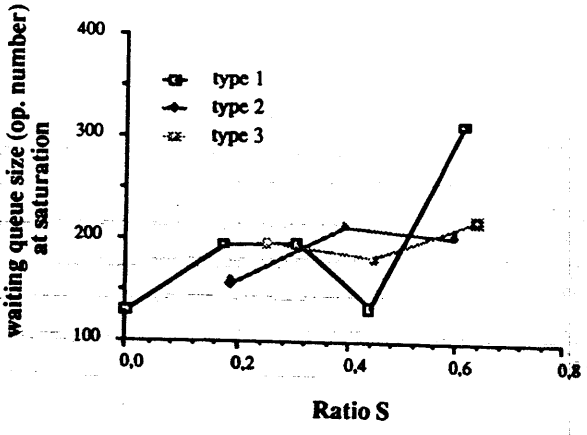


figure 10-a. reservation ratio.

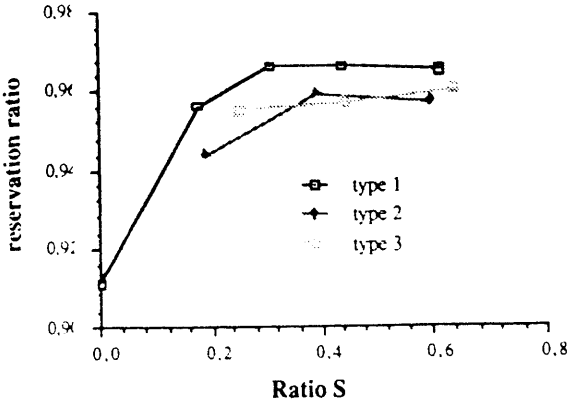
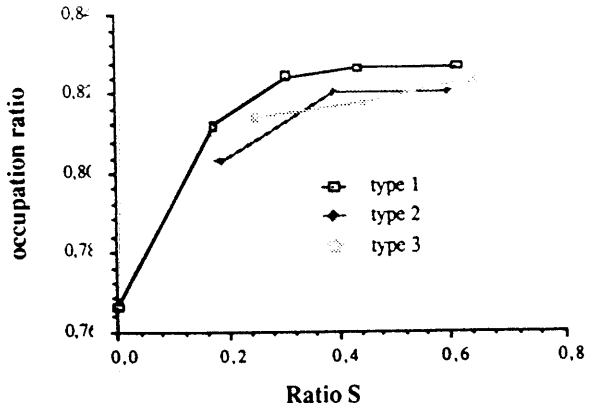


figure 10-b. occupation ratio.



Conclusion

In this paper we have presented a mixed MIMD / SIMD parallel execution model, which is characterized by a non-chaotic operation of the model. Context switches are limited as much as possible, and the ratio of reconfigurations to the total calculus time is also limited. Simulations show that a very good occupation ratio of the host processors seems possible. The model seems to be well adapted to the use of the Rapid associative coprocessor, and to reconfigurable host machines. It appears as an alternative to models based on a very high number of communicating processes, without much synchronization between them. Though the transputers are in principle adapted to an environment of communicating processes, a more organized communication might improve their use. Apart from improving the execution model, hardware and software solutions reducing the reconfiguration durations might also be useful to improve the performances of reconfigurable machines.

In this paper, we have studied the main lines of the model. Several problems have been left open, such as the impact of concurrency control on the compatibility between various operations, the problems of commuting calculus and decoding processes on the same partitions, the effect of inter-clusters operations, or that of operations where data need to be transmitted between processors. These problems need to be studied with a more accurate simulation model, or possibly with a simplified prototype of the software machine. We intend to use these first results as a basis for the implementation of this prototype, which will make possible the use of Rapid on shared nothing multiprocessors. This prospect represents a promising solution, as multiprocessors are rapidly developing and new applications seem to be more realistic in that context. Other works are also currently done on the embedding of Rapid in shared-memory multiprocessors.

Acknowledgements. The Rapid project has been partly financed in 1989 by the Agence Nationale pour la Valorisation de la Recherche (ANVAR). In 1990 it is partly financed by the Ministère de la Recherche et de la Technologie (MRT), within the AASI-89 programme, and by the PRC-AMN.

References

[Abiteboul & Kanellakis, 1989] Abiteboul, S., Kannelakis, P.C., "Object Identity as a Query Primitive", Proc. ACM SIGMOD, Portland, Oregon, Juin 1989

[Boral & DeWitt 1980] Boral H., DeWitt D.J.,"Design Considerations for data-flow database machines." Proc. ACM-SIGMOD 1980 Int. Conf. Management of Data, May 1980, pp. 95-104.

[Boral & Dewitt 1985 a] Boral, H., DeWitt, ed. D., J., "Database Machines", Proc. 5th International Workshop on Database Machines, Grand Bahama Island, March 1985, Springer

[Boral & Faudemay 1989] Boral, H., Faudemay, P., ed. "Database Machines", Proc. 6th International Workshop on Database Machines, Deauville, France, Juin 1989, Springer

[Cheiney & al.1986] Cheiney, J.P., Faudemay, P., Michel, R., Thévenin, J.M., "A Reliable Parallel Backend Using Multi-Attribute Clustering and Select Join Operator", Proc. Int'l Conf. on Very Large Data Bases, Kyoto, Aout 1986

[Chisvin & Duckworth 1989] Chisvin, L., Duckworth, J., "Content-Addressable and Associative Memories", Computer, Juillet 1989

[Eich 1989] Eich, M., "Main Memory Database Research Directions", in [Boral & Faudemay 1989]

[Faudemay 1987] Faudemay, P., Etiemble, D., Bechennec, J.L., He, H., "The Database Processor Rapid", in [Kitsuregawa 1987]

[Faudemay & al. 1988 a] Faudemay, P., Bechennec, J.L., Etiemble, D., "A Highly Parallel Processor with an Instruction Set Including Relational Algebra" (extended abstract), Int. Conference on Computer Design, Port Chester, USA, oct.1988

[Faudemay & al., 1988 b] Faudemay, P., Gardarin, G., Pucheral, P., Thévenin, J.M., "Moyens de verrouillage d'accés pour unité de gestion d'accés en mémoire, et gestion de conflits d'accés utilisant de tels moyens de verrouillage", (in french) european patent pending, application n° 88401820, aug. 19, 1988

[Fushimi & al. 1986] Fushimi, S., Kitsuregawa, M., Tanaka, H., "An Overview of the System Software of a Parallel Relational Database Machine GRACE", Proc. Int'l Conf. on Very Large Data Bases, Kyoto, Aout 1986

[Gajski & al., 1982] Gajski, D., Padua, D., Kuck, D., Kuhn, R., "A Second Opinion on Data-Flow Machines and Languages", Computer, february 1982

[Harp 1987] Harp, J.G., "Phase 2 of the reconfigurable transputer project - P1085", Esprit'87, achievements and impacts, part 1, North Holland, 1987

[Haskin & Hollaar 1983] Haskin, R.L., Hollaar, L.A., "Operational Characteristics of a Hardware Based Pattern Matcher", ACM Trans. on Database Systems, 8, 1, March 1983

[Klinkhamer, 1969] Klinkhamer, J.F., "Memory Addressing Device Using Arbitrary Directed Graph Structure", US Patent, Appl. N° 869 299 (Oct. 22, 1969), Patented Oct. 19, 1971, priority Netherlands.

[De Micheli 1988] De Micheli, G., ed., IEEE Int'l Conf. on Computer Design, Rye Brook, New York, Octobre 1988

[Inmos 1988] Inmos Ltd., "Transputer Reference Manual", 1988, Prentice Hall, NY, USA

[Kitsuregawa 1987] Kitsuregawa, M., ed. "Database and Knowledge Base Machines", Proc. 6th International Workshop on Database Machines, Karuizawa, Japon, Oct.1987, Kluwer

[Kitsuregawa 1989] Kitsuregawa, M., et al., "Implementation of LSI Sort Chip for Bimodal Sort Memory", in [Musgrave 1989]

[Laguna 1988] The Laguna Beach Report, "Future Directions in DBMS Research", Sigmod Record, 18, 1, mars 1989

[Lécluse & Richard, 1989] Lécluse, C., Richard, P., "The O2 Database Programming Language", Proc. Int'l Conf. on Very Large Data Bases, Amsterdam, Aout 1989

[Lee & al., 1987] Lee, C., Su, S.Y.W., Lam, H., "Algorithms for Sorting and Sort-Based Operations Using a Special Function Unit", in [Kitsuregawa 1987]

[Lee & al. 1989] Lee, I., King, R., Paul, R.P., "A Predictable Real-Time Kernel for Distributed Multisensor Systems", Computer, June 1989

[Lee & Herman 1987] Lee, K.C., Herman, G., " A High Performance VLSI Data Filter", in [Kitsuregawa 1987]

[Lee & Mak 1989] Lee, K.C., Mak, V.W., "Design and Analysis of a Parallel VLSI String Search Algorithm", in [Boral & Faudemay 1989]

[Musgrave 1989] Musgrave, G., ed., Proceedings VLSI 89, Int'l Conference, IFIP TC 10 WG 10.5, Munich, août 1989, Elsevier

[Stankovic 1988] Stankovic, J.A., "Misconceptions about Real-Time Computing: A Serious Problem for Next-Generation Systems", Computer, 21, 10, oct. 1988

[Stonebraker & al., 1976] Stonebraker, M.R., et al., "The Design and Implementation of Ingres", ACM Trans. on Database Systems, 1, 3, Sept 1976

[Stonebraker 1989] Stonebraker, M., "Future Trends in Database Systems", IEEE Trans. on Knowledge and Data Engineering, 1, 1, Mars 1989

[Takahashi & al. 1987] Takahashi, K., Yamada, H., Nagai, H., Hirata, M., "Intelligent String Search Processor to Accelerate Text Information Retrieval", in [Kitsuregawa 1987]

[Tanaka 1984] Tanaka, Y., "Bit Sliced Algorithms for Search and Sort", 10th Int'l Conf. on Very Large Data Bases, Singapour, aout 1984

[Tandem 1988] Tandem Performance Group, "A Benchmark of Non-Stop SQL on the Debit-Credit Transaction", Proc ACM SIGMOD 1988, Chicago, Juin 1988