THE UNIVERSITY OF MICHIGAN

COMPUTING RESEARCH LABORATORY[1]

A RELATIONAL DATABASE MACHINE ANALYSIS AND DESIGN

Ghassan Zaki Qadah

CRL-TR-17-83

Under the Direction of
Professor Keki B. Irani

APRIL 1983

Room 1079, East Engineering Building
Ann Arbor, Michigan 48109
USA
Tel: (313) 763-8000

---

[1]This research was supported by the Department of the Army, Ballistic Missile Defense Advanced Technology Center, Rome Air Development Center, and the Defense Mapping Agency under contract F30602-80-C-0173, and by the Air Force Office of Scientific Research/AFSC, United States Air Force under AFOSR contract F49620-82-C-0089. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the funding agency.

ABSTRACT

A RELATIONAL DATABASE MACHINE:

ANALYSIS AND DESIGN

by

Ghassan Zaki Qadah

Chairman: Keki B. Irani

The collection of data in the form of an integrated database is a sound approach to data management. The conventional implementation of the database system, that is, augmenting a large general-purpose von Neumann computer with a large complex software system, the database management system (DBMS), suffers from several disadvantages. Among these are low reliability and poor performance in support of a large class of database operations. The importance of database systems, the disadvantages of its conventional implementation, the advancement of processor-memory technology, and the continuous drop in its fabrication cost inspired a new approach to database system implementation. This approach replaces the general-purpose von Neumann computer with a dedicated machine, the database machine (DBM), tailored for the data processing environment and, in most cases, utilizing parallel processing to support some or all the functions of the DBMS. The new approach claims to improve database system reliability and performance.

The general framework of this thesis is the design of a DBM suitable for supporting concurrent, on-line, very large relational database systems. In designing this machine, a structured approach is followed. First, the relational data model, together with its most important operations and the previously proposed DBMs, is reviewed. This review, coupled with the requirements of the very large database systems and the restrictions imposed by the current and the anticipated state of technology, is used to formulate a set of design guidelines. Consequently, an architecture for a cost-effective DBM that meets this set of guidelines is synthesized. A review of the previously proposed DBMs is carried out using a novel classification scheme. This scheme not only aids one to understand the various organizations of the previously proposed DBMs as well as their design trade-offs and limitations, but also provides a tool to qualitatively analyze and compare DBM effectiveness in supporting the requirements of the very large database systems.

Within the context of the proposed machine, the implementation of a very important relational algebra operation, the equi-join operation, is extensively studied. A large set of algorithms is suggested to implement the equi-join operation on the DBM. An average-value modeling technique is proposed and used to evaluate the equi-join implementations and determine the best performing ones.

Finally, the implementation of the other relational algebra operations as well as other primitives essential to the new DBM are developed.

TABLE OF CONTENT

LIST OF FIGURES

Figure

LIST OF APPENDICES

Appendix

LIST OF TABLES

Table

CHAPTER 1

INTRODUCTION AND BACKGROUND

1.1. Introduction

A conventional database system is one which is implemented by augmenting a large general-purpose conventional von Neumann computer with a large complex software system, the database management system (DBMS). A DBMS is needed primarily to map the user transactions which manipulate the high-level logical data model (the way the data is viewed by the users) into primitives which manipulate the physical data model (the way the data is actually stored). This implementation suffers most from three problems: The first problem is the existence of a complex software system. This reduces the database system reliability and speed. The second problem is the low processing power which can be provided by the underlying Von-Neuman computers and the third problem is the inability of such a system to meet the projected future demands for more processing power. This processing power is needed to support the projected large increases in database sizes, the increase in the database usage, the concurrent user environment, and the important database system features such as data integrity, data security and user views mechanisms.

The limitations of conventional database systems, together with the continuous advancement in memory-processor technology as well as the continuous reduction in its fabrication cost, inspired a new approach to database system implementation; one which uses parallel processing and special function

hardware to support directly the operators manipulating the logical data model. The set of hardware units which is used to support the database system is a dedicated one and is called the "Database Machine." This approach reduces the software complexity because many functions which were implemented by software are now implemented by hardware, thus making the system more reliable and efficient. It also provides the database system with large processing power, which enables it to support a highly concurrent user environment as well as a comprehensive integrity, security, and user views mechanism without degrading the system performance.

Traditionally, the organization of databases has been classified broadly into three types [DATE77], namely, the hierarchical, the network, and the relational models. In the hierarchical and network models, the user views the data as a set of records connected in specific structure (tree for hierarchical and general graph for network). In these models information is represented in several ways, namely, by the *content* of records, by the *connections* between records, by the *ordering* of records, and by the *access paths* defined on the records. On the other hand, the relational model presents the data as a set of two-dimensional tables. In this model, the information is represented by the *content* of such tables.

The relational model has several advantages over the hierarchical and network models. From the implementation point of view, the relational model lends itself well to parallel processing. From the user point of view, the relational model provides for a high degree of data independence and presents the data in a simple and symmetrical way. From the system point of view, the relational model rests on a strong theoretical foundation. This makes the rigorous study of good database system design possible and allows the development of relational database languages as precise as mathematical languages (relational calculus languages).

The general framework of this thesis is the design of a database machine (DBM) suitable for supporting concurrent, on-line, very large, relational database systems. A structured approach is followed to arrive at such a design. First, the relational data model together with its most important operations and the previously proposed DBMs are reviewed. Next, the latter review coupled with the set of the very large database systems requirements and the restrictions imposed by the current and anticipated state of technology are used to formulate a set of design guidelines. Consequently, an architecture for a cost-effective DBM that meets this set of design guidelines is obtained. In the rest of this Chapter, the relational data model and its most important operations are reviewed.

In Chapter 2, the previously proposed DBMs are reviewed. This review is carried out using a novel scheme for the classification of such machines; one which will help us to understand the various organizations of these machines together with their design trade-offs and drawbacks. In Chapter 2, the latter reviews, coupled with both the requirements of the very large database systems as well as the restrictions imposed by the current and anticipated state of technology, are used to formulate a set of design guidelines for the new machine.

Presented in Chapter 3 are the architectural features of a DBM designed to meet the set of guidelines developed in Chapter 2.

In Chapter 4, the implementation of an important relational database operation, the equi-join operation, on the proposed DBM is studied in great detail. A large set of algorithms for such implementation is proposed. An average-value analytical technique for modeling both the proposed algorithms and hardware is used to evaluate the various implementations and determine the best performing ones.

In Chapter 5, a powerful set of algorithms is presented for implementing some other operations for the relational data model together with some primitives essential to the new DBM.

Finally, in Chapter 6, a summary of the work is presented as well as the future directions for the work reported in this thesis. The main contributions of this work to the DBM area are also summarized.

1.2. The Relational Data Model [CODD70, CHAM76, DATE77, TSIC77]

In order to define formally the relational data model, the definitions of the terms relation, key, and normalized relation must be introduced.

(a) Relation

In mathematics, the term relation may be defined as follows:

Definition 2.1 *Relation*

Given a collection of sets $D_1, D_2, \ldots, D_n$ (not necessarily distinct), R is a ***relation*** on these n sets if

$$R \subseteq D_1 X D_2 X \cdots X D_n$$

The above definition implies that a relation is a set of ordered tuples $<d_1, d_2, \ldots, d_n>$ such that $d_i \varepsilon D_i$ for $i = 1, 2, \cdots n$. The sets $D_1, D_2, \ldots, D_n$ are called the ***domains*** of the relation, the positive integer n is called the ***degree*** of the relation and the number of tuples in the relation is called the relation ***cardinality***.

In general, an n-ary relation can be viewed as a two dimensional table with the following properties:

(1) Each row of the table represents an n-ary tuple of of the relation,

(2) The ordering of the rows is not significant,

(3) None of the rows is identical,

(4) The ordering of the table's columns is significant.

In the above representation, it is customary to name each of the table's columns, thus eliminating the tabular representation property number 4. The names of the columns are called ***attributes***.

(b) Key

Definition 2.2 *Key*

A ***key*** (candidate key) of a relation R is a subset of the attributes of R with the following time-independent properties:

(1) In each tuple of R, the value of the key uniquely identifies that tuple.

(2) No attribute in the key can be discarded without destroying property Number 1.

It is possible for a relation to have more than one key. Usually one of these keys is designated as its ***primary key***.

(c) Normalized Relation

In general, relations can be categorized into two subclasses, namely, the normalized relations and the unnormalized relations. A relation is normalized if every attribute of the relation is defined on an atomic (nondecomposable) domain. The representation of relations as normalized relations was introduced primarily for two reasons [DATE77], namely, the normalized relation imposes no real restriction on what can be represented in the database and the normalized relations result in a data structure which can be manipulated

easily by a simple set of operators. Several levels of normalization exist, namely, the first, second and third normal forms (3NF). Successive levels of normalization help the database system to further eliminate some of the undesirable properties related to the data storage operators (insert, delete and update).

At this point, with the previous definitions in mind, the relational data model can be defined as follows:

Definition 2.3 *Relational Data Model*

The relational data model is a collection of time-varying, normalized [usually in the third Normal Form (3NF)] relations of assorted degrees.

1.3. Operations on the Relational Data Model

In general, normalized relations can be manipulated by two classes of languages, the relational calculus and the relational algebra languages. The relational calculus family grew from the observation that the first-order predicate calculus can be used to specify relations which can be derived from the database normalized relations. The relational algebra family rests on a set of relational algebra operators introduced by CODD [CODD70, CODD72] in the early 1970s. The relational algebra operators take relations as operands and yield new relations as a result. While the relational calculus languages are less procedural than the relational algebra, and therefore better for the user interface, the relational algebra languages can be more directly implemented on a parallel machine.

In order to formally define the relational algebra operators, the following definitions must be introduced.

Definition 2.4 *Concatenation*

Given the tuples $r = \langle r_1, r_2, \ldots, r_m \rangle$ and $s = \langle s_1, s_2, \ldots, s_n \rangle$, the *Concatenation* of r with s ($\overline{rs}$) is the $(m+r)$-ary tuple:

$$\langle r_1, r_2, \ldots, r_m, s_1, s_2, \ldots, s_n \rangle$$

Definition 2.5

Let R be an n-ary relation, $r \varepsilon R$, and let $\{A_1, A_2, \ldots, A_n\}$ be the set of attributes of R, then

(1) $r[A_i]$ designates the value of the i^{th} attribute in the tuple r,

and

(2) If $A \subseteq \{A_1, A_2, \ldots, A_n\}$, then

(a) $r[A]$ is a tuple containing only the values, in r, of those attributes specified by A

and (b) $R[A] = \{r[A] : r \varepsilon R\}$

Definition 2.6

Assume that all the attributes in the database have different names and let

AT be the set of all attributes in the database,

$AT(R) \subseteq AT$ be the attributes of the relation R,

DT be the set of all possible values of the domains underlying all the attributes in the database and

$DT(at)$ be the set of all possible values of the domain underlying the attribute at.

Then

(1) A *Predicate* (P), defined on the relation R, is an ordered quadruple $< R, at, \vartheta, v >$ where $at \varepsilon AT(R)$, $\vartheta \varepsilon \{=, \neq, >, <, \geq, \leq\}$ and $v \varepsilon [DT(at) \cup AT(R)]$. If ϑ is " = " and $v \varepsilon DT(at)$, then the corresponding predicate is of the simple type. Otherwise, it is of the complex type.

(2) A *Predicate Conjunction* (PC) is a conjunction of predicates. That is,

$$PC = \bigcap_{j=1}^{m} P_j \text{ where } m \text{ is some finite positive integer.}$$

(3) A *Qualification Expression (QE)* is a disjunction of predicate conjunctions. That is,

$$QE = \bigcup_{i=1}^{n} PC_i \text{ where } n \text{ is some finite positive integer.}$$

Definition 2.7

If the two relations R_1 and R_2 with the respective attribute sets $AT(R_1)$ and $AT(R_2)$ are given, while

$$A \subseteq AT(R_1)$$

and

$$B \subseteq AT(R_2),$$

then A and B are compatible sets of attributes if, and only if,

(a) A and B have 1−1 correspondence, and

(b) If $a \varepsilon A$ and $b \varepsilon B$ and a, b correspond to each other, then a and b are defined on the same domain.

With the previous definitions in mind, the relational algebra operators, namely; select, project and ϑ-join are introduced:

(a) Select

Select is a unary operator. It takes one relation R as an operand, together with some controlling information, the qualification expression QE, to produce a new relation R'. R' is formed from those tuples of R which also satisfy QE. Formally the select operator can be defined as follows:

Definition 2.8 *Select*

$$S_{QE}(R) = \{r : r \varepsilon R \cap QE(r)\}$$

(b) Project

Project is a unary operator. It takes one relation R as an operand, together with a set of attributes $A \subseteq AT(R)$, to produce a new relation R'. R' is formed by selecting the attributes, specified by the set A, from the relation R, then the duplicate rows are removed. Formally, the project operator can be defined as follows:

Definition 2.9 *Project*

$$P\left[R[A]\right] = \left\{r[A] : r \varepsilon R\right\}$$

(c) Θ-Join

ϑ-join is a binary operator. It takes two relations, the source relation S and the target relation T, together with a predicate P, to produce a new relation R. R, which is called the output relation, is formed by concatenating a

tuple of S together with a tuple of T whenever the predicate evaluated on the two tuples is true. Formally, the ϑ-join can be defined as follows:

Definition 2.10 *ϑ-Join*

Let

$\vartheta \varepsilon \{ =, \neq, >, \geq, <, \leq \}$
$A \varepsilon AT(S)$
$B \varepsilon AT(T)$

where A,B are compatible attributes then,

$$J_{A\vartheta B}[S,T] = \{\overline{st}: s \varepsilon S \cap t \varepsilon T \cap (s[A] \vartheta t[B])\}$$

Several operators which are related to the ϑ-join operator are of special importance, namely, the equi-join, the natural-join and the implicit-join. The equi-join is a ϑ-join where the relational operator ϑ is " =."

The natural-join is the same as the equi-join except that the redundant attribute(s) generated by the operation is removed.

The implicit-join is the same as the ϑ-join except that the physical concatenation of the source and the target tuples are not performed. That is, the output relation R is formed of tuples from the target relation T which satisfy the predicate $A\vartheta B$.

1.4. The Database Queries

The general workload of a relational database system consists mostly of a set of queries. A query is a high-level nonprocedural specification of data. In general, the system queries can be grouped into two categories, namely, the retrieval queries and the update queries. A retrieval query gets some data items from (a copy of) the database. On the other hand, an update query per-

forms one of the operations, insertion, deletion or modification. While the first/second operation adds/deletes some data items to/from the database, the third operation updates some data items that already exist in the database.

In general, a retrieval query can be decomposed into a set of relational algebra operations,* such as the selection, the projection and the ϑ-join. The set of operations which corresponds to a retrieval query, can be represented by a tree structure. The nodes in this tree represent the relational algebra operations. The edge(s) that terminates into a node represents the input relation(s) to the corresponding operation. The edge that originates from a node represents the relation which results from processing the operation represented by that node. The leaf nodes of the tree reference only the permanent relations in the database. In most database systems, the leaf nodes of a typical retrieval query are mostly of the selection type.

Similar to the retrieval query, an update operation can be represented by a tree structure (a tree with only the root node). The node, in the latter tree, references only one permanent relation and has one of the operations, insertion, deletion or modification.

*Throughout this thesis, the two words operation and operator will be used interchangeably.

CHAPTER 2

DATABASE MACHINES LITERATURE SURVEY

2.1. A Classification Scheme for the Previously Proposed Database Machines

A database machine (DBM) is a collection of specialized hardware units dedicated and tailored to support some or all of the functions of the database management system (DBMS). The database machines (DBMs) proposed so far have been organized according to the "back-end" concept. The basic idea behind this design concept [CANA74] is shown in Figure 2.1. In a conventional database system all of the system's major software components, namely, the operating system, the database management system (DBMS), the system language(s) translator(s) and optimizer(s), as well as the application programs, are executed on a single general-purpose von Neumann computer which has direct access to the database. On the other hand, in the "back-end" database system, all or part of the DBMS is implemented on a separate machine, the database machine dedicated to support all or part of the DBMS functions, and has exclusive access to the database. The rest of the database system functions are supported by a general-purpose computer (called the host). In this organization the host and the database machine have a master-slave relationship. The host computer, the master, passes high-level access requests to the back-end machine, the slave. When the back-end machine completes the access, it passes the response back to the host. The back-end design concept can be generalized to include configurations where one database machine or more would serve many hosts.

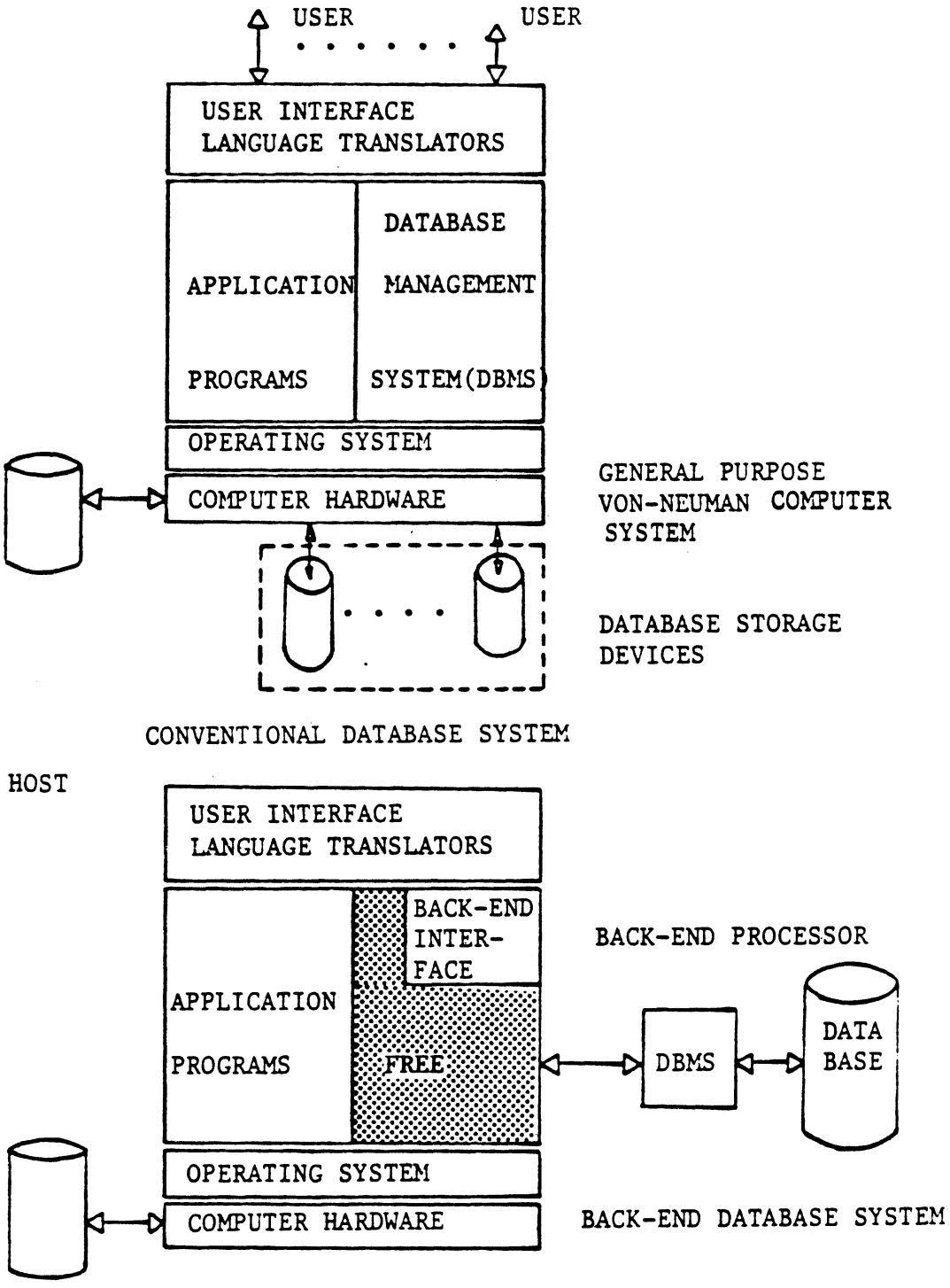

Figure 2.1 Conventional vs Back-End Database System

During the past decade a large number of DBMs have been proposed. Some of them have also been implemented. Others have been commercialized. All of these machines have been designed either to partially or totally support the relational databases* or to support the relational database together with the other database types, namely, the network and the hierarchical databases. In this chapter, the set of DBMs proposed so far is reviewed. This review stresses the way the DBMs are organized as well as the relational database functions** they were designed to support.

A novel scheme to classify the set of DBMs proposed so far is developed. This scheme not only helps us to understand the different organizations of the DBM, together with their design trade-offs and limitations, but also provides us with a useful method of qualitatively comparing the various DBMs proposed. The study participates in formulating some guidelines for designing a cost-effective DBM.

The new classification scheme views the DBMs as points in a three dimensional space, the DBM space. The coordinates of this space, as indicated in Figure 2.2, are: the indexing level, the query processing location and the processor-memory organization.

The most fundamental and important operations the DBMs are designed to support are the selection (from a permanent relation) and the modification operations. In the early designs of the DBMs these operations were carried out using a pure associative approach. That is, the whole database (a set of permanent relations) would be scanned and the data items which satisfy the selection/modification qualification expression would be retrieved/modified. Quickly, researchers in the DBM field came to realize that such an approach is

*The DBMs which were designed to support the file management functions (early DBMs) can also be considered as partially supporting the relational databases since these machines store their files in relation form and support the relational algebra operation selection.

**The most important of these functions, in relational terminology, are the selection, the projection, the θ-join and the modification operations.

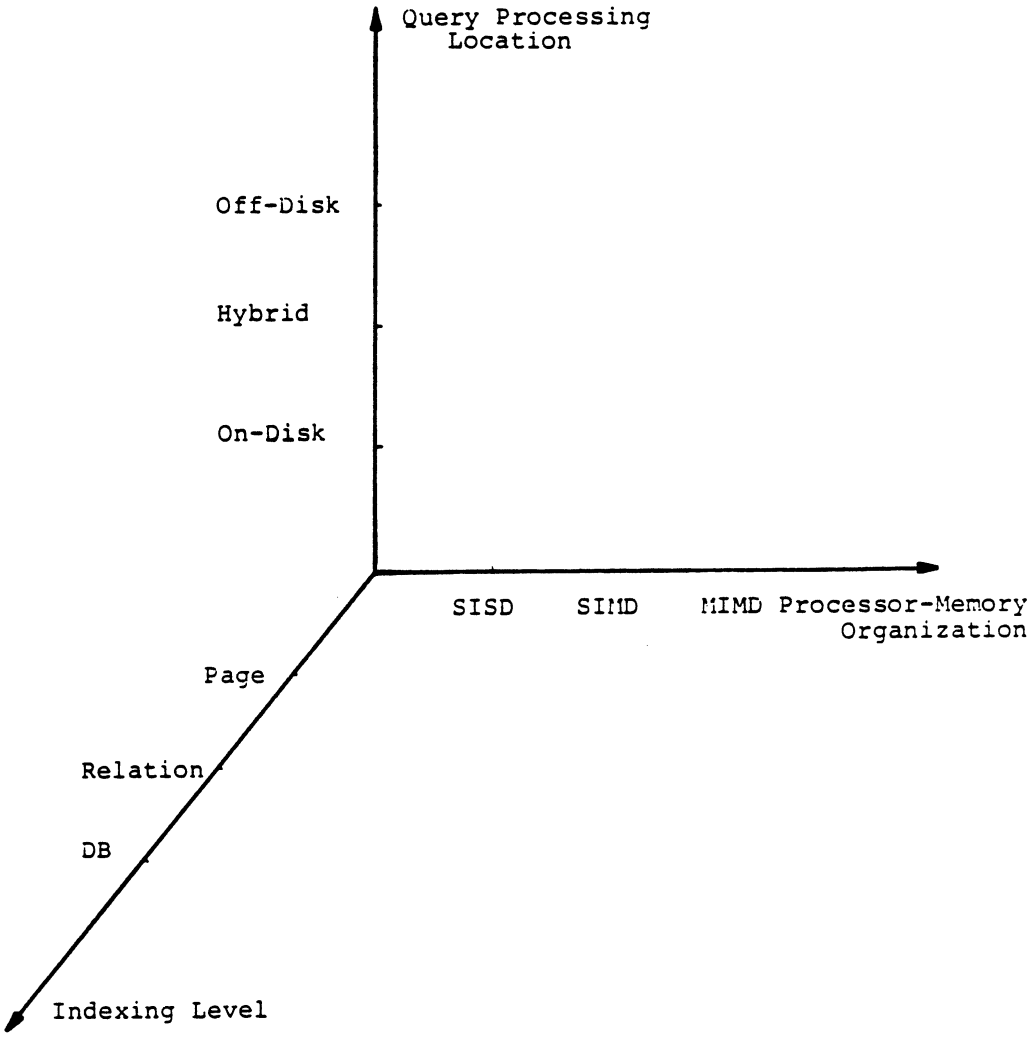

Figure 2.2 The Database Machine Space

not a cost-effective one since the whole database needs to be scanned, at least once, for every selection or modification operation regardless of the size (cardinality) of the operation's output relation.

To achieve a more cost-effective DBM design, a new approach for performing the selection and modification operations has been followed. It is called the quasi-associative approach. In this approach only a relatively small portion of the database (rather than all) needs to be processed for every operation. To support such an approach, the database is divided into a set of data units. In order to perform the selection operation, for example, the machine has first to map the selection qualification expression, to the set of data units which contain (in addition to some other unwanted data) the data which satisfy the qualification expression. Then each data unit of the latter set would be associatively searched to extract the data items which satisfy the selection qualification expression.

In all the DBMs proposed so far, the structure, in use, to map the qualification expression of the selection operation to the data units of the permanent database is the index table [MART77]. These tables are defined for the database and need to be stored and maintained. The data unit, the smallest addressable unit of data, can be logical, namely, the database (no indexing), the relation, or can be physical, namely, a set of tracks, a track or part of a track of a moving/fixed head disk. The physical data unit is called a page.*

The first coordinate in the proposed scheme is the indexing level defined for the permanent database and supported by the particular DBM. Along this coordinate, the DBMs can be grouped into three categories, namely, DBMs with database indexing level, DBMs with relation indexing level and DBMs with page

* Indexing on subtrack pages has been used extensively in the conventional database systems. In the context of DBMs, indexing on relations or relatively large pages (multiple tracks) has been used. This approach reduces the size of the index tables to be maintained, thus reducing their storage and maintenance cost and at the same time improving the system response time and/or throughput.

indexing level. The first category includes all the DBMs which support only the pure associative approach. The DBMs of the second category support the quasi-associative approach. The index tables of the machines, in this category, are defined for the permanent relations as the minimum addressable units.**

The DBMs of the third category support also the quasi-associative search approach. However, in addition to supporting the relational level index tables, they support the index tables which are defined for the pages (containing tuples from the permanent relations) as the minimum addressable units.

The second coordinate in the proposed scheme is the query processing location. Along this coordinate, the DBMs can be grouped into three categories, namely , the off-disk,*** the on-disk and the hybrid categories. The DBMs of the first category execute the query off the disk where the database is stored. In doing that, the DBMs of this category need to move the data relevant to the query from the disk to a separate processor-memory complex where the query processing would take place.

The DBMs of the second category execute the query on the disk. The machines of this category do not need to move data from the disk to a different memory for processing. The disk (a memory) is provided with logic units and the query processing would be carried out on the disk where the database is stored.

The DBMs of the third category execute part of the query on the disk, the selection (from permanent relations) operations and in some machines the

**To facilitate the parallel processing as well as the movement of data, some DBMs of the first/second category store the corresponding minimum addressable unit of data (DB/relation) on a set of physical units called the minimum access units (MACUs), each could be moved separately. However, when a data item needs to be retrieved, all the MACUs containing the DB/relation are processed. In the DBMs of the third category the page (the minimum addressable unit) is stored in one MACU.

***The disk here implies a moving-head disk, a fixed-head disk or an electronic disk, such as the magnetic bubble memory (MBM) or the charge-coupled device memory (CCD). The disk(s) stores the database.

update operations, and move the resulting data to a separate processor-memory complex where the rest of the query execution (if any) would take place.

The third coordinate in the proposed scheme is the processor-memory organization. This coordinate characterizes the hardware of the database machines. For the on-disk/off-disk machines, this coordinate characterizes the way the processor-disk/processor-memory complex executes the database operations. For the hybrid machines this coordinate characterizes the way both the processor-disk and processor-memory complex execute the database operations.

Along the third coordinate, the DBMs can be grouped into three categories, namely, the single instruction stream-single data stream (SISD), the single instruction stream-multiple data stream (SIMD) and the multiple instruction stream-multiple data stream (MIMD) categories. This grouping for the DBMs is similar to that made by Flynn [FLYN72] for the uniprocessor and the multiprocessor systems.

In the following sections, the DBMs proposed so far, together with the corresponding database systems that they are designed to support, are reviewed. This is done within the context of the newly proposed classification scheme.

2.2. The DBMs with the Database Indexing Level

Figure 2.3 shows the database machines which fall within this category. These machines have been organized as an associative processor (AP). An associative memory is a collection or assemblage of elements having data storage capability which are accessed simultaneously and in parallel on the basis of their content rather than by a specific address [KOHO77]. On the other hand, an associative processor (AP) can be defined as an associative memory with

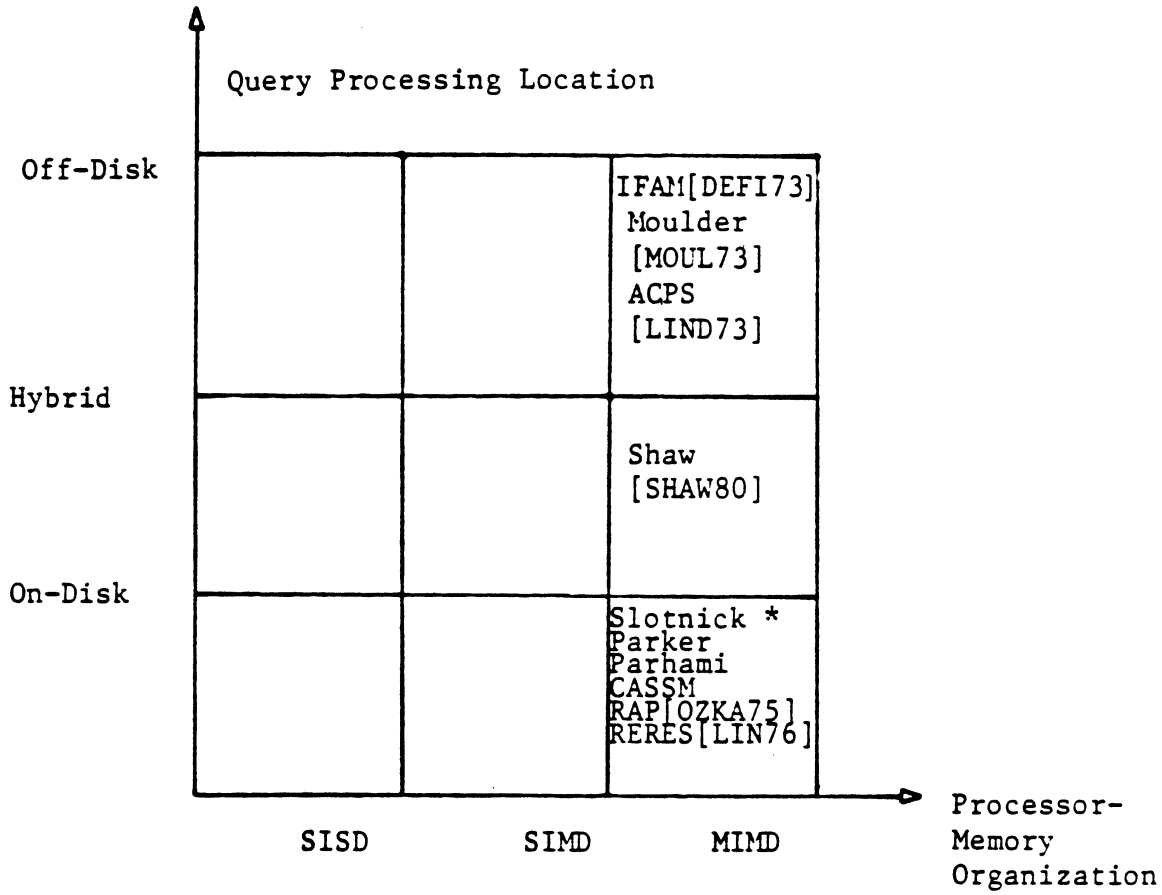

Figure 2.3 The DBMs with DB Indexing Level

*Slotnick[SLOT70],Parker[PARK71],Parhami[PARH72],CASSM[LIPO78]

the added capability of sophisticated data transformation and arithmetic operations.

Many architectures have been proposed for the AP. In [YAU77] these architectures were grouped into four categories, namely: the"fully parallel," the "bit/byte serial-word parallel," the "bit parallel-word serial" and the "block-oriented" categories.

According to the scheme presented in Section 2.1, the DBMs which fall within this category can be grouped further into three subcategories, namely, the off-disk with DB indexing level DBMs (Off-Disk-DB), the on-disk with DB indexing level DBMs (On-Disk-DB) and the hybrid with DB indexing level DBMs (Hybrid-DB). In the following section, the DBMs which fall within the above subcategories are reviewed.

2.2.1. The Off-Disk with DB Indexing Level DBMs

The machines within this subcategory were among the early proposals for a database machine. These machines were provided with the capability of supporting simple file management functions (selection based on simple predicates, simple update, etc.), rather than supporting the functions of a comprehensive database management system.

The database machines of this subcategory are organized around the "bit/byte serial-word parallel" APs. An AP of the latter type is organized as a set of relatively short memory words. Each word is associated with a processing element (simple hardwired logic unit with one bit/byte data path). The processing elements are organized in a synchronous SIMD fashion, where a single hardwired common control unit, executing one instruction at a time, directs and controls their activities. The memory words are implemented using high-speed serial shift registers. The processing elements are physically separated from the memory words. The processing elements execute

parallel search, boolean, update and simple arithmetic operations. In the following sections, the DBMs which fall within this subcategory are reviewed.

2.2.1.1. IFAM Project

DeFiroe et. al [DEFI73] developed a data management system called "information systems for associative memories (IFAM)." An experimental prototype of IFAM was implemented at Rome Air Development Center. This implementation was based on a 2048-word, 48 bits each, bit serial-word parallel associative processor called AM developed by Goodyear.

IFAM stored its data in a simple relational format, called the associative normal form (ANF). The capabilities of IFAM were closely tied to the simple operations which can be performed by the AM. Among these operations were selection based on equality, inequality and within-range match.

2.2.1.2. Moulder Project

Moulder [MOUL73] described the implementation of an experimental, research oriented, hierarchical data management system. The implementation was based on the "bit serial-word parallel" associative processor STARAN and a parallel head-per-track magnetic disk. The system was provided with a high-speed parallel channel to move data between the disk and STARAN.

Using the techniques developed by DeFiore [DEFI71], the hierarchical data were converted into the ANF and stored on the head-per-track disk. The capabilities of the system were close to that of IFAM. The database was partitioned into a number of physical disk sectors which were successively read into the STARAN memory in a high speed parallel fashion, where they were searched using the associative capabilities of STARAN.

2.2.1.3. APCS Project

Linde [LIND73] proposed the "associative processor computer system (APCS)," to support simple data management functions. This system was based on a "byte serial-word parallel" associative processor. Linde conducted a performance comparison of the APCS with a conventional data mangement system implemented on an IBM 370/145 computer system. In that study he hypothesized the existence of a very large random access memory to store the database. The memory is provided with a high speed parallel I/O channel with the capability of transferring 1.6 billion bytes per second to and from the associative processor memory.

Under the above assumptions, Linde found that, in comparison with the conventional system, APCS required less storage and was faster in executing the simple retrieval (simple selection) and update operations.

In general, the database machines presented above suffer from several problems. The most important ones are:

1. The high cost

Organizing a database machine around a "bit/byte serial-word parallel" AP is very expensive. Although the logic-memory cost dropped substantially during the past decade, this AP implementation continued to be very costly. This is due to the fact that such implementation demands a very large amount of logic and fast memory devices. The high cost of this implementation limits the size of the databases that can be supported to relatively small ones.

2. The AP Memory Load/Unload Bottleneck

In general, the size of the database is much larger than the capacity of the AP memory. The database, therefore, is stored on relatively inexpensive secondary storage (disks). The database is partitioned into a number of

data units. Each unit can fit in the AP memory. To search the database, the data units would be successively read into the high speed AP memory where the search would be performed. Unless the system is provided with I/O channels of huge capacity, an I/O bottleneck would be created due to the difference between the search speed of the AP and the I/O channel speed. As a result of the latter bottleneck, the system performance will be considerably degraded. On the other hand providing the system with high-capacity I/O channels increases the system's cost considerably.

2.2.2. The On-Disk with DB Indexing Level DBMs

The database machines of this subcategory were organized around the "block-oriented" APs. An AP of the latter type is organized as a set of memory blocks. Each memory block is associated with a processing element. The memory blocks are made of slow, relatively inexpensive, rotating storage devices, such as the tracks of a fixed-head-per-track magnetic disk or its electronic counterpart such as the charge-coupled memory device (CCDs) [THRE78], or magnetic bubble memory devices (MBM) [CHEN78]. The set of processing elements are organized in an SIMD fashion, controlled by a single control unit. The processing elements are constructed using hardwired logic and are provided with the capabilities of searching and updating the data as it passes by the disk heads. In the following sections, the database machines which fall within this subcategory are reviewed.

2.2.2.1. The Early Proposals

Slotnick [SLOT70] was probably the first to propose adding logic to a rotating storage device. The system he proposed associated logic with each pair of tracks of a magnetic fixed-head disk. The data is read from one track, processed by the head logic and then written back on the other one.

Communication lines were added to permit the tracks to be processed by logic units other than those associated with them.

Parker [PARK71] proposed a system for information retrieval based on a block oriented AP. In Parker's system, every track of a magnetic head-per-track disk is associated with a read head, a write head and a processing element. The data is stored on the disk tracks as variable length records identified by keywords. Under the control of a single control unit, the set of processing elements can in parallel, retrieve, update and count records based on the corresponding keywords.

Parhami [PARH72] proposed a system for information retrieval similar to that of Parker. The proposed system is called RAPID, (rotating associative processor for information dissemination). RAPID was based on a block-oriented AP. RAPID stored its data on a head-per-track magnetic disk in string format. The processing elements of RAPID are provided with simple pattern-matching capabilities as well as capabilities similar to those of Parker's system.

One important note on the above proposals for a DBM is the fact that these systems were designed to support only simple file management functions rather than comprehensive database management systems with their associated logical data models.

2.2.2.2. CASSM

The DBM CASSM* [COPE73, LIPO78, SU73, SU79] was proposed at the University of Florida. It was designed to support the three main data models, the hierarchical, the network and the relational. CASSM is organized as a "block-oriented" AP. The storage media used is a fixed-head-per-track disk. Each track is associated with a read head, a write head and a

*CASSM stands for "context addressed segment sequential memory."

processing element. Each processing element can communicate directly with its adjacent neighbors.

The processing elements, controlled by a single processor, perform the data processing functions in an SIMD fashion. The control processor is responsible for communicating with the host(s) computer(s), distributing instructions to the processing elements, and collecting the final results.

The key feature of CASSM is its "segment sequential" storage scheme introduced by Healy et. al [HEAL72, HEAL76]. A record in this scheme is composed of an arbitrary number of fixed length words. A word is organized as an ordered pair:

<attribute name, value>

The scheme stores the values of the non-numeric attributes only once in the database, separate from the words in which they are values. In these cases the "value" field of the ordered pair is a pointer to the corresponding non-numeric value.

The scheme associates a fixed number of mark bits with each attribute and each record in the database. These are used to identify the result data of one operation that is the input to a subsequent operation. The set of records, constituting the database, are packed together in a file. The file, which can be thought of as a one-dimensional string of words, is subsequently divided into equal length segments, each stored in bit-serial fashion on a track of the head-per-track disk.

When performing the selection operation, the processing elements mark all the records (tuples) which belong to the relation referenced by the operation in one disk revolution. A second disk revolution is used to mark those words of the marked records which satisfy the selection qualification expression. A third disk revolution is used to output the marked words. In

the event that the marked attribute is non-numeric, the third disk revolution is used to check the values of the marked words, pointed by the pointer in the "value" fields. A fourth revolution is needed to output the marked non-numeric values.

CASSM implements only the implicit-join* operation. This operation is implemented using a hashing scheme and an auxiliary random access memory. Since this scheme was first proposed for use in the CAFS [BABB79] DBM, its description is postponed until the latter machine is reviewed.

2.2.2.3. RAP

The DBM RAP** [OZKA75] was proposed at the University of Toronto. It was designed specifically to support the relational data model. RAP, in its original design, is organized as a "block-oriented" AP. The storage media, similar to that of CASSM, is a fixed-head-per-track magnetic disk, each track is associated with a read head, a write head and a processing element. RAP's processing element is a hardwired logic unit designed specifically for non-numeric processing. It is more complex than that of CASSM (for example, it contains several comparators rather than one as in CASSM). The set of processing elements, operating in an SIMD fashion, is controlled by a single control unit similar to that of CASSM. Each processing element has the capability to broadcast part of its data directly to all the other processing elements.

In RAP, data is organized in a relational format which defer sightly from that proposed by CODD [CODD70]. RAP format allows duplicates of tuples to exist in a relation. It also puts an upper limit on the number of attributes a relation can have. The tuples of RAP relations are stored bit wise along a

*For the definition of this operation, the reader is referred to Section 1.3

**RAP stands for "relational associative processor."

track. Only tuples from one relation are allowed on a track. As in CASSM, a tuple is augmented with a fixed number of mark bits (attributes are not), but cannot span two tracks.

Processing of the selection operation in RAP is similar to that in CASSM, however, it is faster because of the simpler storage structure and because the processing element in RAP can perform multiple comparisons (as many as the number of comparators in one processing element) in parallel verses performing one comparison in CASSM.

RAP implements the projection operation using the mark bits. The tuples participating in the projection operation normally are selected (marked) during the execution of an earlier one. RAP executes the projection operation in the following way:

> A processing element of RAP retrieves one marked tuple and resets its marking bit. The processing element forwards the value of projection attributes of the selected tuple to the controller. It also broadcasts the latter to all the processing elements. The processing elements then reset the mark bit of every marked tuple which contains values that match the broadcast one. This process is repeated until all the mark bits are reset.

RAP processes the implicit-join operation as a series of selection operations on the larger relation using the values of the join attribute in the smaller relation as the selection qualification expression.

An analytical performance comparison between RAP and a hypothetical uni-processor DBMS is described in [OZKA77]. The uni-processor is assumed to store the database in a sequential ordered form with indices defined for all the attributes of the database. The performance study shows that RAP was between three and sixty times faster than the DBMS in executing the selection operations. When comparing the execution of the modification operation, RAP was found to be 5,000 times faster than the DBMS. This is due to the extra efforts associated with updating the indices by the DBMS. For the more complex operations, such as the implicit-join or the projection,

RAP shows no significant performance advantage over the uni-processor DBMS.

Since its introduction in 1975, RAP underwent many design changes. The new design of RAP (called RAP.2) falls within the off-disk-relation DBM category. RAP.2 is reviewed later within the context of its own category.

2.2.2.4. RARES

RARES* [LIN76] was designed at the University of Utah as an intelligent controller for a head-per-track magnetic disk. It is provided with only the capability to perform the selection operation on the relational data of the disk. RARES stores the relational data in a different way from that of RAP. Tuples in RARES are stored across the tracks in byte-parallel fashion (in RAP tuples are stored along the tracks in a bit-serial fashion). The set of tracks used to store a tuple is called a band. The band sizes vary according to the tuple length.

RARES hardware consists of a set of specially designed search units (hardwired logic) with relatively wide datapaths (64 bits). Each search unit is associated with 256 tracks. These tracks are partitioned into four groups. The search unit has the capability to search the tracks within any of these groups in parallel.

Several advantages have been claimed for RARES orthogonal storage layout over that of RAP-like systems. The most important one is that in outputting the selected tuples, contention, likely to occur in RAP-like systems, between the processing elements for the I/O bus can be reduced. On the other hand, this organization is not without drawbacks. It does not distribute the workload evenly. For the selection operation where the qualification expression involves one attribute, only one search unit is busy, the

*RARES stands for "rotating associative relational store".

others simply wait for a command to transmit their portion of the tuple. In addition, close communication between all the search units is required during the output of a selected tuple since all the search units must output its share of the tuple in specific sequence. This is needed in order to allow the controller to reconstruct the selected tuple.

In general, the database machines which fall within the On-Disk-DB category have offered a good solution to the data movement problem. By moving the logic to data rather than moving the data to logic, the need for the costly data movement has been eliminated. On the other hand, this solution is not without its own drawbacks. The most important ones are:

1. Low Speed

The On-Disk-DB machines process the query on relatively slow rotating storage devices. In a performance report [OZKA77], RAP (and RAP-like processors) are shown to be effective in processing the selection and modification operations which require few disk revolutions. RAP on the other hand is very slow (as slow as the conventional DBMs) in executing the more complex operations, such as the projection and the ϑ-join operations. This is due to the fact that such operations are implemented* as repeated search operations with a fixed long rotational time for each of these search operations regardless of the number of tuples needed to be searched.

2. High Cost

Associating logic with each head of a head-per-track disk is an expensive approach to DBM design. This is due to two factors, namely, the high cost of the head-per-track disks as mass storage media (at least one order of magnitude more expensive than the moving-head-disk storage technology) and the high cost of associating logic with every track of the disk. Both of

*One exception to this implementation is that of CASSM.

the previous factors cause these DBMs to have expensive mass storage media, thus limiting them to supporting only small databases.

2.2.3. The Hybrid with DB Indexing Level DBMs

Only one DBM, the Shaw machine [SHAW80], falls within this category. The Shaw DBM is designed specifically to support the relational data model. It is organized as a two-level hierarchy of APs. At the top of the hierarchy is the primary associative memory (PAM). PAM is fairly fast, small AP, organized as "bit/byte serial-word parallel." At the bottom of the hierarchy is the secondary associative memory (SAM), a large, relatively slow AP. SAM is organized as a "block-oriented" AP (the same as RAP). Each of the blocks processing elements have the additional capability of performing some nontrivial arithmetic operations such as multiplication and division. PAM and SAM are provided with a channel of adequate bandwidth for intercommunication.

SAM stores the relational database using a storage structure similar to that of RAP. The Shaw machine executes the selection operation on permanent relations in SAM in the same way that RAP does.

The Shaw machine implements the equi-join operation. The algorithm in use partitions the Join attribute underlying domain into disjoint sets such that the expected number of the source and target tuples which map to any of these sets would fit in PAM's local store. The sets of tuples, one set at a time, would be retrieved by SAM and transferred to PAM. PAM then would join the sets, one at a time, using an algorithm similar to that of RAP.

One method for domain partitioning is the hashing function technique. A hashing function, computed by the track processing elements of SAM, would select those tuples of the source and target relations which hash to one partition of the join domain and forward them to PAM where the rest of the equi-

join operation would be concluded.

The Shaw machine also implements the projection operation. The algorithm in use exploits the domain partitioning techniques similar to that in use for the equi-join operation.

The performance of the Shaw machine is superior to that of both the Off-Disk-DB and the On-Disk-DB. In comparison with the Off-Disk-DB DBMs, the Shaw machine eliminates the need to move the whole database to the fast AP (as a matter of fact, only a small portion of the database needs to be moved). Thus relatively inexpensive I/O channels can be used without any serious I/O bottlenecks. In comparison with the On-Disk-DB DBMs, the Shaw machine moves the execution of the complex relational operations from the slow rotational device, the head-per-track disk, to a much faster device, the "bit/byte serial-word parallel" AP.

The Shaw machine is not without problems. One of the most important ones is the high cost of organization. The Shaw machine combines two very expensive technologies: the head-per-track disk (obsolete as mass storage media) with nontrivial logic associated with each head and the "bit/byte serial-word parallel" AP. The cost factor restricts the Shaw machine to support only the relatively small database systems.

2.3. The DBMs with Relation Indexing Level

Figure 2.4 shows the database machines which fall within this category. According to the scheme presented in Section 2.1, these DBMs can be grouped into three subcategories, namely, the off-disk with relation indexing level (off-disk-relation), the on-disk with relation indexing level (on-disk-relation) and the hybrid with relation indexing level (hybrid-relation). In the following sections, the DBMs within the latter subcategories are reviewed.

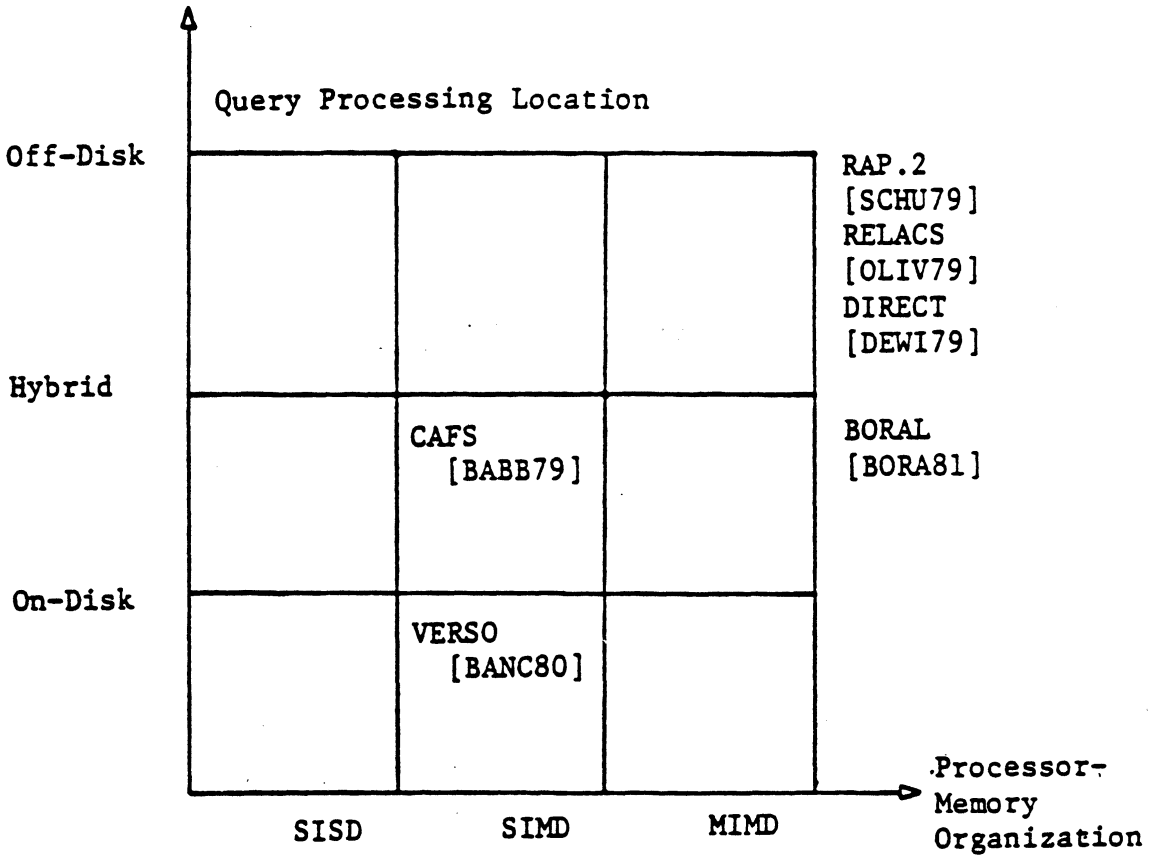

Figure 2.4 The DBMs with Relation Indexing Level

2.3.1. The Off-Disk with Relation Indexing Level DBMs

2.3.1.1. RAP.2

RAP.2 [SCHU79] is a modified version of the DBM RAP. RAP.2 consists of a number of processing elements, each with a pair of tracks. The set of processing elements, implemented as microprocessors, are controlled and managed by a single controller. The controller, implemented as a mini/micro processor, and the set of processing elements process data in an MIMD mode. The processing elements can intercommunicate indirectly through the controller. In RAP.2 the track is implemented using one of the new memory technologies, such as the charge-coupled device memory (CCDs), the magnetic bubble memories (MBMs) or the electron-beam addressable memories (EBAMs).

RAP.2 stores the database on some number of conventional mass storage devices (moving-head disks, for example). For every relation in the database, the controller keeps a list of the secondary storage track addresses which store the given relation. Before processing any relational operator, the controller brings all the tracks which store the relations, referenced by the given operator from the secondary storage, to the tracks of RAP.2. The processing of the relational operator then proceeds in the same way as that of RAP. Providing two tracks to every processing element in RAP.2 is needed to overlap the I/O data movement with data processing.

One last note about RAP.2 is the fact that the execution of the relational operators which reference a relation(s) larger than the capacity of the tracks associated with the RAP.2 processing elements were never presented.

2.3.1.2. RELACS

RELACS* [OLIV80] is a DBM proposed at Syracuse University to support large relational databases. RELACS is designed around two APs organized in a "bit serial-word parallel" fashion. The design tries to overcome the weakness of early systems with similar organization (Off-Disk-DB DBMs), namely, the I/O bottleneck due to relatively slow I/O with respect to faster search time and the requirement that the entire database be searched at least once for each selection operation.

RELACS consists of six main functional units: the global control unit (GCU); the data dictionary unit (DDU); the associative units (AU0,AU1); the mass storage device (MSD); and the output buffer (OB). The GCU interfaces and controls the activities of the different functional units of RELACS. The MSD is organized as a set of moving-head disks augmented with a number of high-speed buffer memory modules each with the capacity of one track. The data, organized as relations, are stored on the moving-head disks.

For every relation in the database the DDU stores the list of the track addresses which store the given relation. AU0 and AU1 are extremely fast APs which, in cooperation with the GCU, can execute the relational operations in a limited MIMD fashion. Loading an AP is performed by a custom-designed I/O device which is capable of selecting a module from the buffer memory and transferring its content to the AP's memory. While one buffer memory module is being emptied into an AP, one or more of the other modules can be loaded from the disks.

RELACS processes the selection operation in a simple way. The set of tracks which store the relation referenced by such an operation are loaded into the memory of AU0 and/or AU1, one track at a time. The data then is

*RELACS stands for "relational associative computer system"

searched using the associative capability of the corresponding AP(s).

RELACS implements the ϑ-join operation. The algorithm in use performs selection repeatedly. Part of the source relation is loaded into the AP(s) memory(ies). Then each tuple of the target relation is compared using the associative capability of the AP(s) to the loaded source tuples. The source tuples which respond to a target tuple, together with the target tuple are transferred to the output buffer OB were they are physically joined. This process is repeated until all the source and target tuples have been processed. The implementation of the projection operator on RELACS was never presented.

Oliver [OLIV79] compared the performance of RELACS to RAP in performing the selection, deletion, modification and insertion operations. It was shown that RELACS is faster in almost every case with the exception of the mass addition to the database. While in the worst case RELACS performs marginally better than RAP does, in the best case it is three orders of magnitude faster. In the above, worst and best case refer to the capacity of the AP's memory.

2.3.1.3. DIRECT

DIRECT [DEWI79] is a DBM proposed at the University of Wisconsin/Madison to support large relational databases. It is organized as a set of general purpose microprocessors (termed query processors) whose function is to execute operations, such as selection, ϑ-join and update, on the database. These microprocessors are controlled and managed by a minicomputer [termed back-end controller (BEC)]. The BEC is also responsible for interfacing DIRECT to the outside world [the host(s)], distributing instructions to the query processors and overseeing the data transfers from the secondary memory to the processors. The BEC and the query

processors are organized in an MIMD fashion.

The database in DIRECT, organized as relations, resides on some number of mass storage devices (moving-head disks). Each relation is organized as a vector of fixed-size pages. For each relation the BEC keeps a list of addresses for those pages which store that relation. A number of CCD memory units serve as a fast buffer memory between the query processors and the mass storage devices. The CCD memory units also serve as temporary storage units for the result pages of one operation that are to be used in a subsequent operation. The CCD memory units are managed by the BEC.

The query processors and the CCD memory units are interfaced by a simple cross-point switch that has two important capabilities: any number of query processors can read the same CCD memory unit simultaneously and any two query processors can read from any two CCD memory units concurrently. The cross-point switch also interfaces the mass storage devices to the CCD memory units.

The organization of DIRECT facilitates two types of concurrent query processing, namely, intra-query processing (simultaneous execution of two or more instructions from the same query) and inter-query processing (simultaneous execution of two or more instructions from different queries). Facilitating both types of concurrent query processing are essential for supporting contemporary and future concurrent large database systems.

To execute the selection operation the BEC selects a number of query processors and broadcasts to them the selection operation code and some other needed information. The BEC also brings the relation (referenced by the selection operation) into the CCD memory units. Each query processor participating in the Selection, requests a data page from the controller each

time it is through with its current one. The controller replies with the CCD memory unit address which stores the corresponding data page. The query processor then reads the data page into its internal memory and retrieves those tuples which satisfy the selection criteria using the sequential scan of the data.

DIRECT implements the ϑ-join operation using a parallel version of the nested loop algorithm. In DIRECT, the larger of the two relations being joined is designed as the outer relation, the other is the inner one. Each query processor participating in the ϑ-join receives one page of the outer relation. If the page is not sorted on the joining attribute, the query processor sorts it. Next, the pages of the inner relation, which are sorted on the joining attribute, are broadcast, one at a time, to all the query processors with an outer page. Each query processor joins its outer page with the incoming stream of inner pages. Whenever a query processor's output buffer fills up, the query processor sorts it on the attribute that is to be used in the subsequent operation (if any) and then outputs it to an empty CCD memory unit (whose address is supplied by the BEC). This procedure is repeated until all the pages of the outer relation have been processed.

DIRECT implements the projection operation. The description of the algorithm in use can be found in [BORA81].

Although the organization of DIRECT facilitates concurrent query processing, it is not without problems. In a simulation study for DIRECT [BORA81], it was evident that the BEC is a system bottleneck even for a small number of query processors. This is due to the fact that the organization of DIRECT necessitates that any query processor request for reading/writing any CCD memory unit must pass through the BEC.

The other problem of DIRECT is related to the crosspoint switch. This switch requires that the complexity of the logic at the interface of the CCD memory units (whose number is m) and the query processors (whose number is n) grow as (m x n). Thus, for large n and m this interface would introduce a large delay in the data transfer between the CCD memory units and the query processors. The large amount of logic together with the fact that every CCD memory unit must be provided with enough power to drive all the query processors make the crosspoint switch prohibitively expensive for supporting a large number of query processors.

2.3.2. The On-Disk with Relation Indexing Level DBMs

Only one DBM falls within this category, namely, the VERSO [BANC80] DBM. It is organized as an SISD machine. The database is stored on a number of moving-head disks (conventional or modified for parallel read out from the whole disk cylinder) in relational format. A single processor, which is designed primarily to execute the selection operation, is placed between the disks and the memory device to which the selected data is to be delivered. The processor, which acts as an I/O filter, is organized as a finite state machine which executes very simple microcode instructions and has the capability to scan the data as fast as the disk delivers it.

In [BANC80] it is shown that, if the data is organized in a new normal form, VERSO can be used to perform the equi-join operation with some occasional performance penalty. The new normal form simplifies the join algorithm as well as reduces the number of joins to be performed on the database.

2.3.3. The Hybrid with Relation Indexing Level DBMs

2.3.3.1. CAFS

CAFS* [COUL72, BABB79] is a DBM designed specially for supporting the relational databases and is available commercially from ICL Ltd. In CAFS the database, organized in the relational form, is stored on a number of moving-head disks. CAFS contains several specially designed hardware units, among them are the associative search unit (ASU), the file correlation unit (FCU), and the record retrieval unit (RRU). The first two units process data in a pipeline fashion and at the same rate as the disk delivers it. In both units the data is processed in an SISD fashion.

The ASU is responsible for executing the selection operation. It acts as a data filter to the FCU which is responsible for performing the other complex relational operators. The ASU implements the selection operation as follows:

> The tuples of the referenced relation (referenced by the selection operation) stream through the ASU at the disk transfer rate. The ASU evaluates the selection qualification expression on each passing tuple and marks those which satisfy it. The RRU then collects the marked tuples for output to the user.

The FCU provides an efficient mechanism for the evaluation of the implicit-join and the duplicate removal part of the projection operations. It consists of a number of 1 bit wide random access memories (1-bit-wide vectors). The 1-bit-wide vectors are used to store, in coded form, the set of values present in some join or projection attribute(s) of a relation(s).

The implicit-join operation references two relations, the source (S) and the target (T). In implementing this operation CAFS uses a novel algorithm (it is also used by other DBMs such as CASSM). In this algorithm, the join attribute value of each source tuple is transformed into an index, using a

*CAFS stands for "content addressable file store."

hashing function, of the 1-bit vector. The corresponding bit of the vector is set to "1." Then for every tuple of the target relation, the join attribute value is transformed into an index using the same hashing function. The index is used to access a vector location and the tuple is retrieved if the location value is "1."

Using the above algorithm, some of the target tuples are retrieved without having a match among the values present in the join attribute of the source relation. This is due to the collision phenomenon associated with the hashing functions [KNUT73]. That is, with the hashing functions in use, there is a non zero probability that a given bit will be set by more than one of the values present in the join attribute of the source relation tuples. This probability, as shown in [BABB79], can be made arbitrarily small by the use of a sufficient number of 1-bit vectors together with an equal number of statistically independent hashing functions. However this probability will never be zero and occasionally some target tuples will be retrieved in error.

CAFS implements the duplicate removal part of the projection operation using an algorithm similar to that of the implicit-join operation. The details of the algorithm can be found in [BABB79]. However, the usage of the hashing scheme will result in the loss of some tuples of the projected relation.

Although CAFS implements some novel algorithms for evaluating the implicit-join and the projection operations, it still suffers from many problems. One problem is the fact that the novel algorithms introduce some error in performing the implicit-join and the projection operations. Another problem is the fact that CAFS implements only the implicit-join algorithm efficiently, but not the more general join operations such as the equi-join and the ϑ-join. A third problem is the fact that both the on- and off-disk processing units of CAFS are organized in an SISD fashion with little ability to

support a concurrent query processing environment.

2.3.3.2. Boral Machine

Boral [BORA81] at the University of Wisconsin/Madison proposed a DBM designed specifically to support large concurrent relational database systems. This machine contains a master controller (MC), a set of instruction controllers (ICs), a set of instruction processors (IPs) and a set of modified moving-head disks. These components intercommunicate over a broadband, coaxial cable, broadcast bus that uses frequency-multiplexed, RF-modulated channels to allow for several simultaneous communications over the single bus.

The moving-head disks store the database in relational form. Every head of a moving-head disk is provided with a processing element. The set of processing elements of one disk is controlled by a single controller. For every relation on a moving-head disk, the corresponding disk controller keeps a list of the page addresses which store such relations. The processors of a disk have the capability to perform, in an SIMD mode, the selection and some simple update operations on the data of the tracks as fast as the disk delivers it. The set of moving-head disks can execute instructions in an MIMD fashion.

The MC handles all the communications with the host computer(s), initiates the execution of the instructions, and controls and manages the other machine components.

An IC is a processing element provided by some amount of fast memory and a small secondary store (small moving-head disk). An IC together with a number of both IPs and moving-head disks form what is called an IC group. The different units of an IC group communicate over one channel of the multichannel coaxial cable. An IC, grouped with a set of IP's and moving-head

disks, is assigned an instruction (selection, ϑ-join, projection or any combination of them) for execution. The group IC is responsible for the management of the other components within the group. The number of IPs allocated for the execution of a given instruction is determined by the MC, taking into consideration several factors such as the statistics about the data, etc. The broadband coaxial cable allows a number of IC groups executing different instructions to proceed concurrently.

Boral's DBM executes the selection, the ϑ-join and the projection operations using algorithms similar to those of the DIRECT machine. The details of these algorithms can be found in [BORA81]. In Boral's DBM, the selection operation on permanent relations is executed on the moving-head disks. the result of this operation is staged to the local store of the IPs and ICs where the more complex operations are executed.

Although, Boral's DBM uses what seems to be an "exotic" interconnection scheme, it still suffers from many problems. The first one is the fact that each hardware unit connected to the coaxial cable needs to have an RF-multiplexer/demultiplexer unit. This increases the cost of these hardware units considerably and thus the overall machine cost. Another problem is the fact that the communication of messages over such media requires some form of acknowledgment [MAGL80] performed by the software. This tends to reduce the bus effective transmission rate considerably.

2.4. The DBMs with the Page Indexing Level

Figure 2.5 shows the DBMs which fall within this category. According to the scheme presented in Section 2.1, these DBMs can be grouped into three subcategories, namely, the off-disk with page indexing level (off-disk-page), the on-disk with page indexing level (on-disk-page) and the hybrid with page

indexing level (hybrid-page). In the following sections, the DBMs within these subcategories are reviewed.

2.4.1. The Off-Disk with Page Indexing Level DBMs

2.4.1.1. The Intelligent Database Machine

The intelligent database machine (IDM) [EPST80A, EPST80B] is designed to support the small to medium relational database systems and is commercially available from Britton-Lee, Inc. The database is organized as a vector of small size (subtrack) pages. Indices are defined for the most frequently used attributes of the database. An attribute index contains a list for every value of the corresponding attribute. The list contains the addresses of all the pages which store the tuples having such attribute and value. The IDM stores the database together with the indexing information on up to 16 moving-head disks.

The other important hardware components of the IDM are the master processor, the database accelerator and a cache random access memory with maximum size of 3 Mbytes. The cache is used to buffer the most frequently referenced pages of the database. The master processor is a conventional one which performs some database functions as well as controls the activities of the rest of the machine components. The database accelerator is a custom-designed processor that is claimed to run at speeds ten times the transfer rate of the disk. The accelerator is designed to effectively execute some small portions of the relational DBMS code. It was observed that most of the execution time of the DBMS is typically spent in such code.

The details about the way the IDM implements the different relational operations are not available and thus are not presented here.

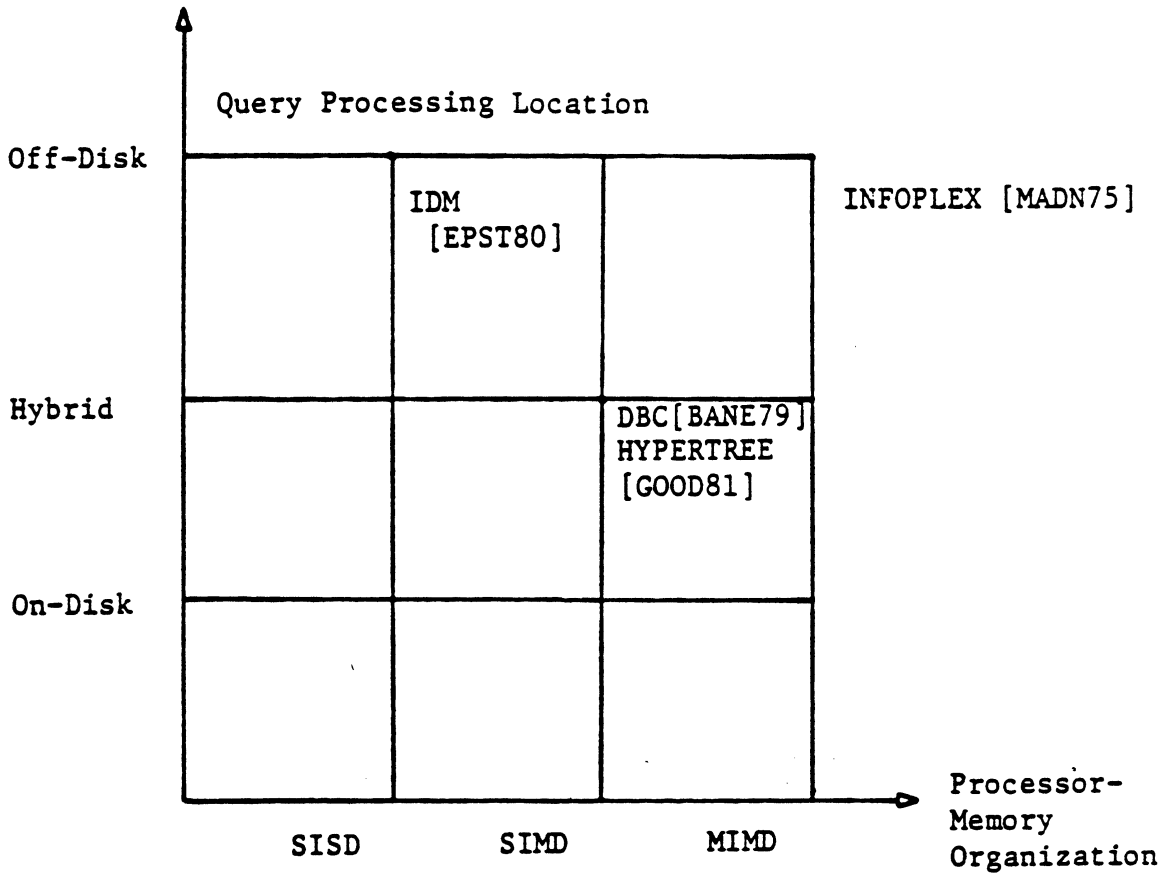

Figure 2.5 The DBMs with Page Indexing Level

2.4.1.2. INFOPLEX

INFOPLEX [MADN75, MADN79] is a DBM designed in a structured top-down fashion. The database system functions were decomposed into a functional hierarchy. Each functional level of the hierarchy is implemented in terms of the functions (primitives) provided by the next lower level. By associating a microprocessor complex to each functional level (the microprocessor complex is optimized to execute the primitives of that level) and piping the data between the levels, both the inter and intra instruction parallelism can be attained. An intelligent storage hierarchy employing different technologies of varying speed and cost is used to store the database. The design of the organizations is based on locality of reference observations in databases.

The details about the implementation of both the functional and the storage hierarchies as well as the implementation of the different relational operations are not available and thus are not present here.

2.4.2. The Hybrid with Page Indexing Level DBMs

2.4.2.1. Hybertree

Hybertree [GOOD80] is a DBM proposed for supporting relational database systems. It is an interesting machine because of the way it was designed. The performance of a number of different interconnection strategies between a number of processing units was examined for the execution of the duplicate removal part of the projection operation [GOOD80]. The various strategies were characterized in terms of their merits and demerits. An augmented physical binary tree structure was picked as the best. In this scheme, processors are organized as a binary tree, but with each node connected in a regular manner to one of its siblings. The leaf

nodes are interconnected using the perfect shuffle structure [STON71] and are connected to a moving-head disk modified for parallel read out of a whole cylinder. The leaf nodes are responsible for the execution of the simple search operations. Thus they act as data filters to the higher level nodes which are responsible for executing the complex relational operations.

The hybertree DBM implements both the projection and the equi-join operations. The duplicate elimination part of the projection operation is implemented using the perfect shuffle or the binary tree connection. The equi-join is implemented using a parallel version of the CAFS hashing algorithm. Each leaf processor initializes, in parallel with the others, its own 1-bit-wide vector with the number of entries equal to twice the number of unique values in the join attribute. Each leaf processor sets up its own 1-bit vector according to the values of the join attribute of the source relation. Parent nodes in the tree are responsible for "ORing" the bit vectors that their children produce. This procedure is repeated for the target relation. The root of the tree receives the final two vectors, encodes the two relations and ANDs them to form a new vector. This vector is broadcast to all the leaf nodes. The source and target tuples are hashed again to the 1-bit-wide vector. A tuple of the source or the target relations is then forwarded to a prespecified processor if it hashes to a "set" bit of the 1-bit-wide vector. The prespecified processors are picked in such a way that tuples that have the same Join attribute value are sent to the same processor. The prespecified processors then join the corresponding assigned tuples.

2.4.2.2. DBC

The DBC [BANE79,BAUM76] is a DBM designed to support very large database systems. The database is organized as a vector of pages (each has the size of a cylinder). Indices are defined for the database in the same way

as those of the IDM. However, since the page size in the DBC is much larger than that of the IDM, the size of the index in the former machine is much smaller than that in the latter.

The DBC consists of seven functionally specialized units. These units are organized in two loops, namely, the data loop and the structure loop. In addition to other units, the data loop contains the mass memory (MM) and the post processing unit. The structure loop contains the structure memory components.

The MM [KANN78, HSIA76B] stores the database on a set of moving-head disks modified for parallel readout of a whole cylinder. The set of disks are connected via a switch to a set of processors (the track processors). The track processors perform the selection as well as the update operations. Thus they act as data filters to the other components of the DBC.

The post processing unit is responsible for executing the equi-join operation. The post processing unit consists of a number of processors interconnected by a uni-directional ring and a single controlling processor that has a communication line to each processor [HSIA79]. In executing the equi-join operation, each processor receives two blocks of tuples, one from the source relation and the other from the target relation. For every tuple of a target block, the corresponding processor joins it with the tuples of the source blocks then communicates the tuple via the ring networks, to its neighbor processor. This is repeated until all the target tuples have been joined with the tuples of all the source blocks. One problem with the post processing unit implementation of the equi-join operation is that it is assumed that the data to be operated on will fit in the memories of all the processors.

The structure memory (SM) [HSIA76A] stores the index information. The SM is to be constructed out of one of the new memory technologies, such as MBMs, CCDs or EBAMs. By storing the database on the moving-head disk such that the most frequently referenced part is clustered together in as few cylinders as possible, the SM can help reduce the number of cylinders which need to be scanned by the track processors for each selection or update operation.

2.5. A Critical Look at the Previously Proposed DBMs

In the previous sections, a comprehensive and detailed survey for the previously proposed DBMs was presented. This presentation was guided by a newly developed classification scheme. This scheme is based on three attributes, namely, the indexing level, the query processing location and the processor-memory organization. In this section, this scheme is used to discuss critically the various DBMs designs. It will also help to highlight the most basic tradeoffs in the design of the various DBMs.

Historically speaking, the first DBMs to be proposed were organized as off-disk machines and were provided with only the associative access to the database (Off-Disk-DB DBMs). In general, these machines (and this design approach) suffered from many drawbacks, in particular, its ineffectiveness in handling the very large database systems. A DBM of the latter type must move all of the database from the slow, rotating mechanical disks where it is stored to a fast (associative) memory where the execution of the selection or the update operations take place. In the large database system environment (database size >> associative memory capacity), the I/O channels easily become the system bottleneck as a result of the mass data movement (all the database) and the high search speed of the associative memory. These drawbacks, coupled with the high cost of producing associative memory units of

large capacities and providing the DBM with wide I/O channels, make this design approach cost ineffective.

The On-Disk-DB design eliminates the data movement problem. The latter design approach achieves that by processing the database operations on the secondary store devices where the database resides. Although this approach eliminates the need to move large volumes of data and thus eliminate the I/O bottleneck, it continues to have many drawbacks. The most important ones are:

1. The above design for DBMs stores the database on a set of logic-per-track disks. Using the latter disk as a mass storage device is very costly, some orders of magnitude more expensive than the moving-head disk. In general, the logic-per-track disk has two components, namely, a fixed-head-per-track magnetic/electronic disk and a set of logic units. Each of these units is associated with the head of the head-per-track disk. The high cost of the logic-per-track disk is attributed mainly to two factors[KANN78], namely, the high cost of the head-per-track disk as a mass storage media and the large number of the logic units associated with such a disk. Currently, the magnetic head-per-track disk is considered to be obsolete as mass storage media. The electronic disk (the MBM devices and the CCD memory) is at least one order of magnitude more expensive than that of the moving-head disk. The future directions in the mass storage technology shows that the electronic disk technology will not challenge the speed/cost level of the moving-head-disk for the near future[HSIA81].

2. The above design for DBMs requires that the logic unit associated with each head of the head-per-track disk be fast enough to perform the selection and update operations on the disk data "on the fly." That is, at the

rate the disk delivers its data. This requirement is hard to satisfy because of two factors, namely, the large variations in the rate of the data processing requirements and the recent improvement in the disk transfer rates (for example, a recently introduced IBM 3380 moving-head disk has a data transfer rate per head of 3 Mbytes/sec [IBM80]).

The above problem is a complicated one and its solution at the hardware level, such as providing the disk with a specially designed, highly fast logic, will largely increase the mass storage (even for the storage system which contains only moving-head-disk units) cost. To appreciate how fast the head logic must be, it must be remembered that the current widespread use of the moving-head disks is attributed to the introduction of high-speed head controllers, implemented using the expensive bipolar logic technology which has the ability to check and correct disk errors on the fly.

3. Assuming the above two problems can be solved, the On-Disk-DB DBMs perform reasonably well in executing the simple database operations which require few disk revolutions (the selection and update operations with simple qualification expressions). On the other hand, they perform poorly in executing the more complex database operations that require many disk revolutions (for example, the ϑ-join and the projection operations). The latter observation was evident in the performance evaluation of RAP[OZKA77] (an example for the On-Disk-DB DBMs).

Recall that a query can be thought of as a tree whose nodes represent a set of database operations. The leaves of the tree reference only permanent relations of the database. In a real database environment, the leaf nodes are mostly of the selection and update type. A hybrid DBM processes the leaf selection operations and, in some machines, the update operations on the disk. The result relations are then moved to a fast processor-memory

complex where the rest of the query operations (if any) are executed. In most cases executing the selection and update operations on the disk largely reduces the volume of data need to be moved to the fast processor-memory complex.

From the preceding discussion, one can draw the following two important conclusions regarding the DBMs of the Hybrid-DB group, namely:

1. The performance of the Hybrid-DB DBMs is superior to that of both the Off-Disk-DB and the On-Disk-DB DBMs. This is due to the fact that the DBMs of the former type execute the database operations on more tailored hardware units and at the same time reduce the volume of data to be moved out of the secondary disks. On the other hand, the Hybrid-DB approach compound the cost problem and do not offer any solution to the problem of "on the fly" processing. The cost problem has been compounded because the hybrid-DB DBMs combine two expensive technologies, namely, the logic-per-track disks and the associative memories.

2. In the light of the current and near future technology, providing the DBM with only associative access to the database will result in a cost-ineffective design. In order to achieve an acceptable levels of performance, the latter approach provides the DBM with very expensive (and will continue to be expensive in at least the near future) mass storage media and logic units.

In general, the number of tuples to be selected/modified as a result of executing a typical selection/modification operation is relatively small in comparison with that of the database. In the light of current processor-memory technology, it is clear that the need to scan the whole database, at least once, to carry out these operations is very cost ineffective. The design approach

which provides the DBM with a mechanism that eliminates the need to scan the whole database will, in general, be more cost-effective.

In the context of DBMs, index tables defined for the permanent database have been used as a mechanism to reduce the amount of data to be processed for a given selection or modification operation. In the scheme presented earlier, the DBMs, which use indexing, have been classified, according to the indexing level they support, into two groups, namely, the DBMs with relation indexing level (DBMs-Relation) and the DBMs with page indexing level (DBMs-Page).

To facilitate data sharing as well as data movement, the DBMs of the DBMs-Relation-type store the database relations on the secondary storage in a set of physical units, each can be moved separately (for example, RAP.2 and RELACE use the disk track as a data unit; DIRECT and BORAL's machine use a page 16 Kbytes long as a data unit). However, whenever an operation references a data item, all the data units which store the relation containing such item are processed.

The index table defined for DBMs-Relation DBM (relation-index) store, for every relation of the database, the set of addresses of the data units which store the corresponding relation. In general, the size of the relation-index index is very small relative to the size of the database. Its maintenance and storage cost are negligible relative to those of the database. To execute a selection/modification operation on an on-disk-relation/hybrid-relation DBM, only the data units which store the relation referenced by the operation will be searched/modified. For those machines which use a logic-per-track disk as storage media, only the tracks which correspond to the former units will be processed. The rest of the tracks can be processed, simultaneously and in parallel, against one or more other selection and/or modification operations.

On the other hand, to execute these operations on an off-disk-relation DBM, only the data units which store the relation referenced by the operation need to be moved to the processor-memory complex for processing.

From the preceding discussion, one can draw the following important conclusions regarding the DBMs-Relation DBMs:

1. In the context of the databases dominated by relatively small size relations, providing a DBM with a Relation-Index index substantially improves its cost-effectiveness. For the On-Disk/Hybrid organized DBMs, the Logic-Per-Track disk allows more than one operation to be processed in parallel or may be replaced by a less expensive mass storage media. For the Off-Disk DBMs the amount of data need to be moved out of the secondary store is substantially reduced. Thus a less expensive I/O channels can be utilized.

2. In the context of the databases dominated by relatively large size relations (in the very large database systems a relation could have the size of several magnetic disks), providing a DBM with a relation-index index does not improve its cost-effectiveness. It is very clear that such DBMs suffer from the same problems as those of the DBMs-DB type.

A DBM of the DBMs-Page group stores the database relations, on the secondary stores, in a set of physical data units, each is called a page [or a minimum access unit(MACU)] and can be accessed and moved separately. The index tables supported by the DBMs-Page DBMs (page-index) is defined on the set of the most frequently referenced attributes of the database. For every value of the latter attribute the page-index index stores the set of addresses of all the Pages which contain tuples having such value. Although the size of the page-index index(relative to that of the database) is a function of the size of

the page itself, nevertheless, the index size as well as its storage and maintenance cost is substantial. Another problem with using the Page-Index index is the need for a clustering mechanism[BANE79]. This mechanism is used to store the set of tuples which are frequently referenced together in as small a number of pages as possible. The use of the latter mechanism will generally improve the performance of the selection and possibly the modification operations. On the other hand, it will introduce some overhead in executing the insertion operations.

In the context of very large database systems, the use of small size page in conjunction with a DBM will result in a relatively large page-index index which requires a huge amount of storage and high maintenance and updating costs. The performance of the update operations will be severely degraded. The improvement in the execution time for the selection operation will not be enough to compensate for the overhead and the latter loss in performance (one example of this approach is the conventional database system which use the von Neumann computer as a DBM). On the other hand, the use of large size pages, in conjunction with a DBM, will result in a relatively small page-index index(~ 1% of the total database size[BANE79]). Thus its storage and maintenance costs will be substantially low and the performance of the modification and selection operations will, in general, be enhanced especially if the page is processed by a number of processors in parallel.

Using the page-index index coupled with large size pages has allowed designers to replace the expensive logic-per-track disk with the relatively inexpensive moving-head disk (or a slightly modified version of it) as a unit for mass storage.

In the scheme presented earlier, the DBMs proposed so far were organized as SISD, SIMD and MIMD machines. In general, the execution of a database

operation,* on a DBM of the first/second group is done serially. That is, one operation (possibly two for the hybrid DBMs) is the maximum number of operations that a DBM of this type can execute at any given time. While the database operation is executed serially (that is, by one processor) on the SISD DBMs, it is executed in parallel (that is, by more than one processor) on the SIMD DBMs. The MIMD DBMs, on the other hand, execute one or more database operations in parallel fashion. The operation itself is also executed in parallel.

In the context of the relatively low cost of the processor and memory devices and the very large database systems, employing parallel processing largely enhances the effectiveness of the DBMs. This was evident in the DBMs presented earlier. Although the SIMD organization of the DBMs largely enhances the execution time of a database operation, it does not offer a real solution to the database concurrent user problem. The MIMD organization is more effective in database systems where fast and concurrent access to the database is a basic requirement. This is due to the fact that the MIMD organization has the ability not only to execute a database operation in parallel but also to execute more than one operation (from same or different queries) simultaneously and in parallel.

One important drawback in the MIMD organization is the overhead in controlling the execution of the different queries and the management of the various system components. In most MIMD DBMs such overhead puts an upper limit on the number of queries or resources that can be active simultaneously in the system. This overhead caused the DIRECT machine to have poor performance[BORA81]. Therefore, controlling and minimizing such overhead must be a basic objective for the MIMD DBM designer.

---

*With database operation is meant the selection, projection ,ϑ-join, insertion, deletion or modification.

2.6. General Guidelines for the Proposed DBM

The general framework of this research is the design of a back-end DBM capable of supporting very large, concurrent relational databases with high performance. This machine is to be called the "relational database machine (RDBM)."

The most important characteristics of the contemporary and future very large database systems are the vast amount of data in such systems and the large number of users requiring simultaneous access to this data. To support such a system, our proposed RDBM must support the latter features in a cost-effective way. This places the following two important requirements on the RDBM, namely:

1. Availability of large capacity storage.
2. Handling the on-line concurrent access to the database with adequate response time and throughput.

To ensure the satisfaction, in a cost-effective way, of the above two requirements, our proposed RDBM will be organized along a set of guidelines. These guidelines have been drawn from our study of the earlier proposals for the DBM as well as the current and future state of technology. These guidelines are:

1. The mass storage in the proposed RDBM is to consist of the moving-head disks. The latter disk type is selected for its ability to provide a vast amount of on-line storage at a relatively low cost and moderate performance. Currently, the magnetic fixed head-per-track disk is considered obsolete as a mass storage device. The electronic disk (the MBM and the CCD memory devices) technology is at least one order of magnitude more expensive than that of the moving-head disk. A look at the future directions in mass storage technology shows that electronic disk technology

will not challenge the speed/cost level of the moving-head disk for, at least, the near future[HSIA81].

2. Supporting the page level indexing. This type of indexing will greatly improve the execution time of the selection and modification operations. On the other hand, it will increase the execution time of the other update operations and introduce some overhead in the form of index table access delay as well as in storing and maintaining the corresponding indices. To minimize the drop in performance due to the overhead associated with indices maintenance, the page must be selected to have large size (multiple tracks of the moving-head-disk) and be processed, in parallel, by a number of processors. Also, it must provide a support to access the page index at the hardware level.

3. Organizing the proposed RDBM as an off-disk DBM. Although this organization introduces some increases in the execution time of the database operations (due to moving the data to the processor-memory complex), it nevertheless avoids providing the moving-head disk with a large amount of specially designed logic and memory devices needed for "on the fly" processing of the selection and update operations. Thus keeping the mass storage cost at a minimum. This organization must also try to improve the execution speed of the latter operations by taking advantage of the locality of references to the database. The processor-memory complex must be designed to effectively support not only the relational algebra operations, namely, the selection, projection and ϑ-join, but also the primitives that manipulate the page index.

4. Organizing the proposed RDBM as an MIMD DBM. This is very important in order to provide the proposed RDBM with the ability to handle concurrent

access to the database. The proposed design must be able to handle the excessive overhead associated with the MIMD DBM organization.

CHAPTER 3

THE RELATIONAL DATABASE SYSTEM ORGANIZATION

In this chapter, the architecture of a back-end DBM suitable for supporting concurrent, on-line, very large relational databases is presented. The newly proposed DBM has been designed to meet the set of the previously stated guidelines (see Section 2.6). However, before presenting such an architecture, the way the data is organized in the new system is outlined and discussed.

3.1. The Data Organization

In general, the proposed relational database system stores two types of data, namely:

(1) The Database

The database is organized as a collection of time-varying normalized relations of assorted degrees. The database is divided into a set of large data units. Each, called the page [or the minimum addressable unit* (MAU)] represents the smallest addressable unit of data. The only tuples which are allowed in the same MAU are those of the same relation. This is done for the following two reasons:

(a) Space Saving

Placing the tuples of different relations in different MAUs will save a considerable amount of storage. This saving is due to the fact that the name of a

*In the newly proposed machine, the minimum addressable unit is contained in one minimum access unit as is seen later.

relation and its tuple format do not need to be attached to each tuple of the database. On the other hand, this organization will result in wasting some storage space due to MAU internal fragmentation. The internal fragmentation would result from having small relations compared to the size of an MAU. The storage waste due to the internal fragmentation is minimal in very large databases since a typical relation in such a database occupies many MAUs.

(b) *Improving The Machine Performance*

In most database environments, there exists a high probability that tuples of the same relation will be referenced together. Placing tuples of the same relation in an MAU would reduce the number of MAUs which need to be accessed and processed for a user transaction, thus improving the machine performance.

(2) *The Database Directory*

The database directory contains the information needed to map a "data name" to the set of MAU addresses which store the named data. In the proposed system, data is named at two levels, namely, the relation level (relation-name) and the tuple level (tuple-name: < relation name, attribute name, value>). While there is one unique name for a relation, more than one name can be attached to one tuple. The number of possible names for a tuple does not exceed the number of the tuple's attributes.

The database directory consists of two indices, namely the relation index and the MAU index. The relation index maps the relation-name to a set of MAU addresses. These MAUs contain all the tuples of the relation whose name is relation-name. The MAU index maps a tuple-name to a set of MAU addresses. Each of these MAUs contains at least one tuple which has the tuple-name as its name. In the following, both indices will be formally defined.

Let

$R = \{r_1, r_2, \ldots, r_l\}$

be the set of relations which form the database;

$AT = \{at_1, at_2, \ldots, at_j\}$

be the set of attributes in the database;

$AT(r) \subseteq AT$

be the set of attributes associated with the relation r;

$V = \{v_1, v_2, \ldots, v_n\}$

be the set of attributes values in the database;

$V(r.at) \subseteq V$

be the set of values in the database associated with

the attribute at of the relation r;

$ADDR = \{addr_1, addr_2, \ldots, addr_k\}$

be the set of all MAU addresses;

$ADDR(r) \subseteq ADDR$

be the set of all MAU addresses which contain the tuples of the relation r.

Definition 3.1 *Relation Index*

The relation index is a set of ordered pairs. Each has the form

$$<r, ADDR(r)>$$

where $r \varepsilon R$.

The MAU index (Figure 3.1) is organized as a *three level* index. The *first level* is the MAU master index (MIND), the *second level* is the attribute index (AIND) and the *third level* is the index-term index (ITI). The index-term index maps a tuple-name to a set of MAU addresses. Each of these MAUs contains at

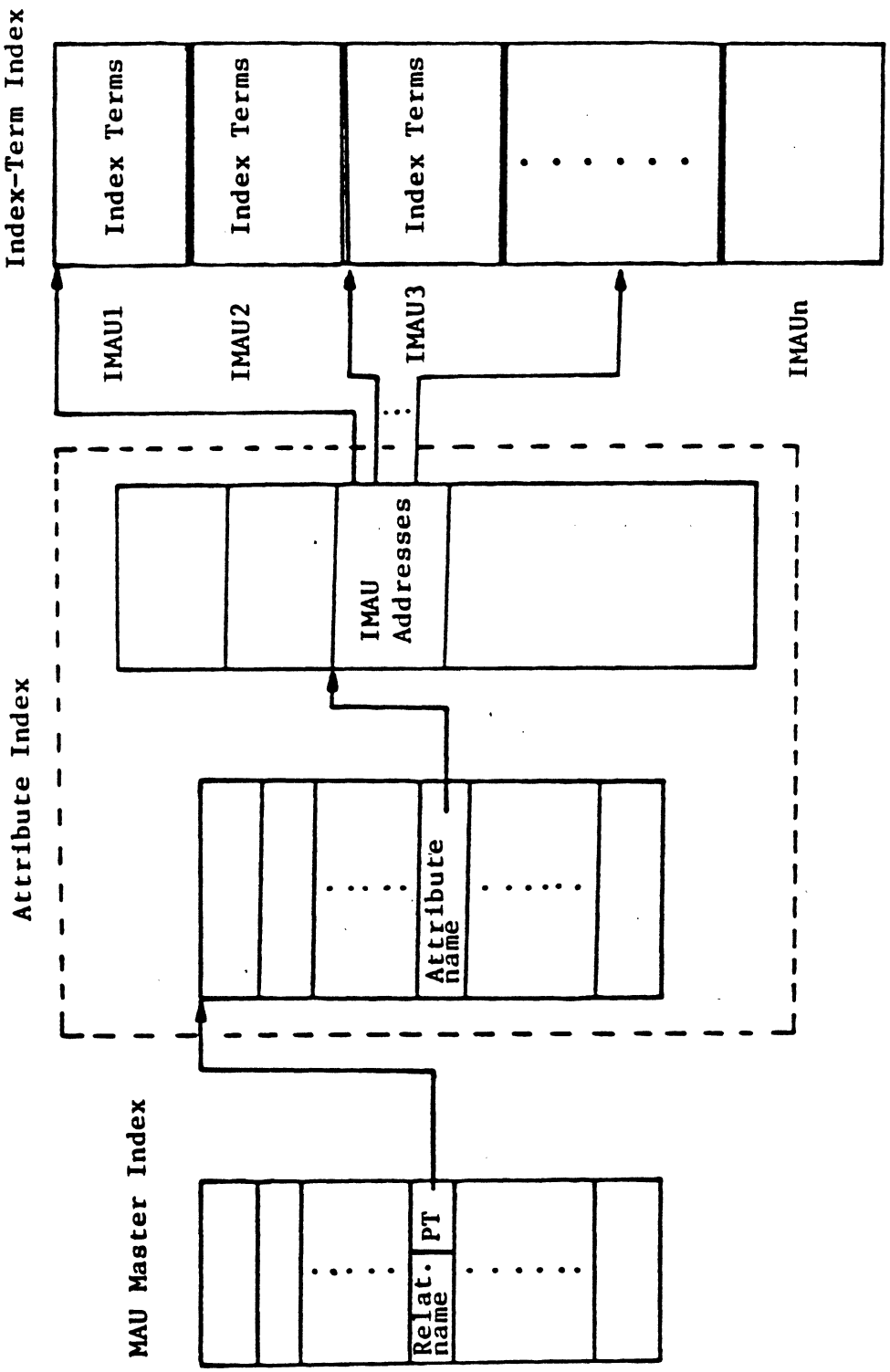

Figure 3.1 The organization of the MAU Index

least one tuple with the above name. The index-term index is a set of index terms. Formally, an index term can be defined as follows:

Definition 3.2 *IndexTerm*

An index-term is an ordered quadruple $<r, at, v, MAUA>$ where $r \varepsilon R$, $at \varepsilon AT(r)$, $v \ \varepsilon V(r.at)$ and $MAUA \ \varepsilon \ ADDR(r)$. The MAUA is the address of an MAU which contains at least one tuple whose name is $<r, at, v>$.

In general, the index terms will be defined only for those attributes which are frequently referenced by users. The index terms in the proposed system are grouped and stored in units, equal in size to an MAU, called the index MAUs (IMAUs). Although an IMAU may contain index terms defined for different attributes of different relations, some clustering mechanism will be used to cluster, into the same IMAU, those index terms which are defined for the attributes of the same relation. This will improve the storage cost as well as the processing efficiency of the index terms.

The MAU master index and the attribute index are introduced in order to reduce the number of IMAUs which need to be processed for a selection or modification operation. The MAU master index maps a relation name to its attributes. The attribute index maps an attribute name to a set of IMAU addresses. The IMAUs contain the set of index terms which are defined for the corresponding attribute. Formally, the two indices can be defined as follows:

Definition 3.3 *MAU Master Index*

The MAU master index is a set of ordered pairs of the form

$$<r, PT>$$

where $r \varepsilon R$ and PT is a pointer to the *second level* index, the attribute index.

Definition 3.4 *Attribute Index*

The attribute index is a set of tables, one table for each $r \varepsilon R$. Each table is a set of ordered pairs of the form, $<at, \{addr_1, addr_2, \ldots, addr_s\}>$, where $at \varepsilon AT(r.at)$ and $\{addr_1, addr_2, \ldots, addr_s\} \subseteq ADDR$. $\{addr_1, addr_2, \ldots, addr_s\}$ is the set of IMAU addresses which store the index terms of the attribute at .

Before leaving this section, an important operator, the *index-select*, which manipulates the index-term index, must be outlined. The index-select operator is executed in conjunction with the relational algebra operator select. It limits the search space of the latter operator to those MAUs which contain tuples that satisfy the corresponding select qualification expression QE.

3.2. The Relational Database Machine (RDBM) Organization

The proposed relational database machine (RDBM), shown in Figure 3.2, may be broken down into four subsystems, namely, the master back-end controller (MBC), the processing clusters subsystem (PCS), the mass Storage subsystem (MSS) and the interconnection network subsystem (INS). In the following sections the architecture of these subsystems is outlined.

3.2.1. The Master Back-End Controller Subsystem

The master back-end controller (MBC) [possibly in cooperation with the front-end computer system(s)] performs the following functions: (1) Interfacing the users to the database system; (2) translating the user queries into the primitives of the processing clusters subsystem; (3) scheduling and monitoring the query execution; (4) storing and maintaining the system dictionary; (5) storing, maintaining and manipulating part of the database directory (the relation index, the MAU master index and the attribute index) and (6) providing for security checking, integrity maintenance and user views. In the fol-

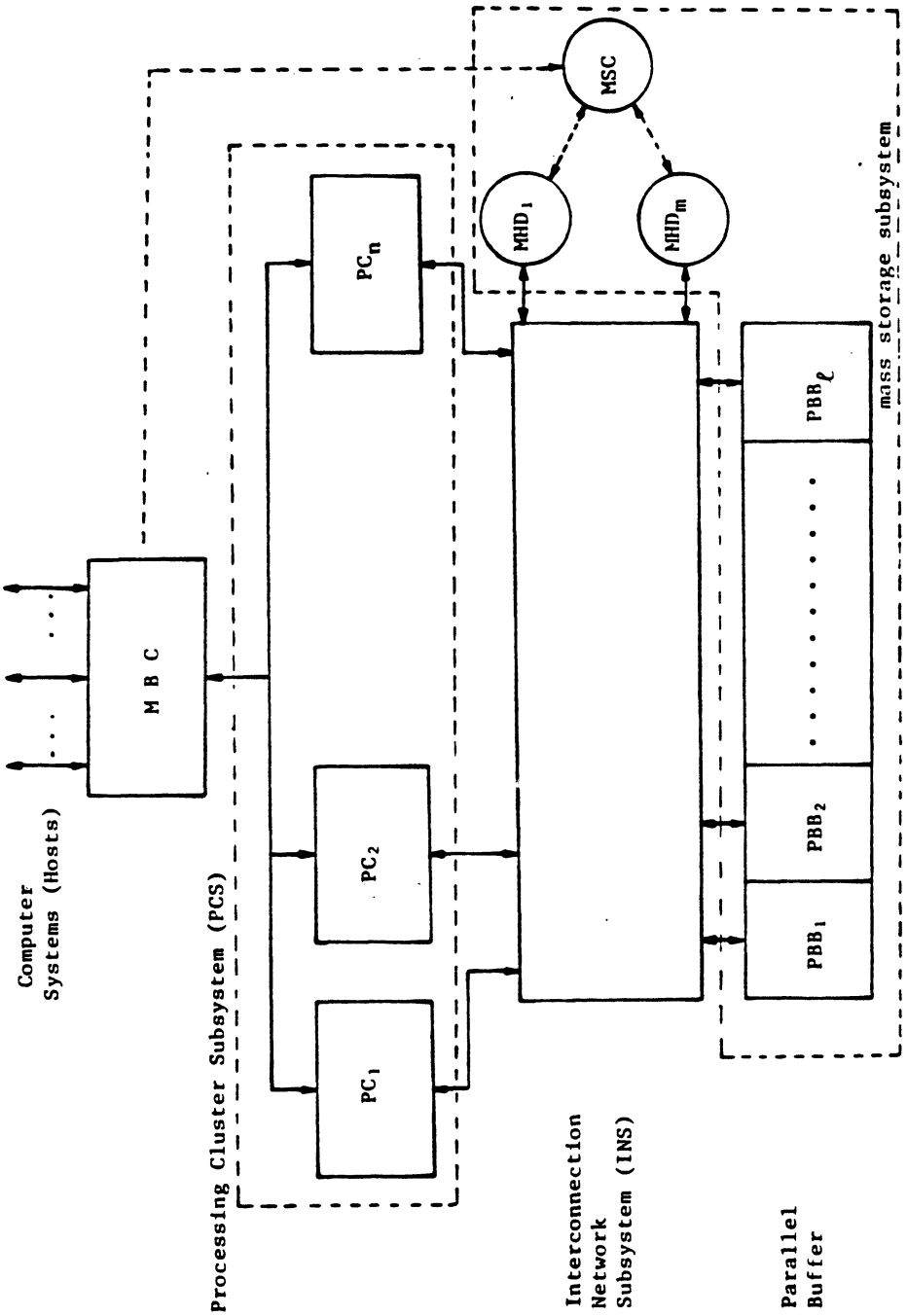

Figure 3.2 The Organization of the New RDBM

lowing paragraphs, the most important functional features of the ones stated above are discussed in more detail.

(a) Maintenance of the System Dictionary

The system dictionary stores all the information which relates to the definition of the database. For each relation in the database, the system dictionary stores its name, cardinality, and the name of its attributes as well as the attribute size, type, etc. The system dictionary also stores the system's users definitions together with their database accessing privileges as well as the users' view definitions.

The information stored in the system dictionary is very important to the relational system. It is used in almost all phases of query processing, namely, query translation, optimization and execution.

(b) Translating and Optimizing the User Queries

Query translation and optimization are essential functions in the database system. In general, the translation process is needed to transform the user query into a set of primitives executable by the system hardware. Whereas the optimization process is needed to find an optimum plan (optimum with respect to certain performance measures) to execute the user query on a given hardware.

In RDBM, a query,* before it is ready for execution, passes through three distinct phases. During the first phase, the query is translated into an equivalent set of relational algebra operations. In the second phase, the execution order of the relational algebra operations, which constitute the query, is rearranged. The primary objective of such rearrangement is to minimize the volume of data which must be manipulated during the construction of the query response set. Also during this phase all the operations, in the

*Throughout the rest of this thesis, the word query refers to the retrieval ones only.

translated query, are assigned methods by which they find their relevant sets of MAUs. For the operations which reference temporary relations (relations obtained from the database permanent relations by applying one or more relational operators), as well as the Θ-join and the projection operations which reference permanent relations, the set of the relevant MAUs are those which store the temporary and permanent relations, respectively. The addresses of such MAUs are obtained from the relation index. For the selection operations, which manipulate permanent relations, a decision must be made regarding the index to be used (the relation index or the MAU index). The decision is based on several factors, namely, the expected size of the operation response set, the localization of such set, and the availability, in the select qualification expression, of simple predicates which reference indexed attributes. If the qualification expression contains simple predicates of indexed attributes, the response set is relatively small and it is localized (span a small number of MAUs relative to the large number of MAUs spanned by the permanent relation), then the MAU index will be used, otherwise the relation index will be used. For a given selection operation, If the relation index is to be used, then the RDBM will process the set of MAUs which store the tuples of the referenced relation. On the other hand, if the MAU index is to be used ,then only a subset of the latter set will be processed. This subset is determined by executing the index-select operation on those IMAUs which contain index terms relevant to the selection operation.

In general, the type of index to be used in association with a selection operation is determined by processing its qualification expression QE. This is carried out in two steps, namely, the QE processing step and the MQE processing step. In the following sections both of these phases are outlined.

(1) *QE Processing*: QE is the qualification expression associated with the selection operation. During this step, the QE is transformed into a modified

qualification expression MQE which has the same form as QE but consists only of predicates of the simple type and is defined on indexed attributes. This step is processed as follows:

> Scan QE, one predicate at a time. Whenever there exists a predicate of the complex type or a predicate of the simple type but defined on an unindexed attribute and the predicate is the only remaining member of the corresponding predicate conjunction, then the rest of this phase is skipped and MQE is assigned the value null. On the other hand, if the predicate is not the only remaining member, then it is deleted and the scanning continues. The QE remaining after this processing is the MQE.

When the processing of the QE is finished, the result is the modified qualification expression MQE which is either null or contains simple predicates defined on indexed attributes. If the MQE is null, then the relation index is used and the MQE processing step is skipped. On the other hand, If the MQE is not null, then, with the aid of the system dictionary, the ratio of the number of MAUs which store the tuples satisfying MQE and the number of MAUs which store the whole referenced relation, is estimated. If this ratio is relatively small, then the MQE processing step is carried out, otherwise, the relation index is used.

(2) *MQE Processing*: In this step, the set of IMAU addresses which contain the index terms relevant to MQE is found. This is done as follows:

> Replace every predicate in MQE with the set of IMAU addresses which contain index terms of the corresponding attribute. The set of IMAU addresses corresponding to every predicate conjunction in MQE is then found by intersecting the sets corresponding to every predicate of a predicate conjunction. Then, the set of IMAU addresses which contain the index terms relevant to MQE is found by taking the union of the sets which correspond to every predicate conjunction in MQE.

At the end of the second step, both the MQE, the set of IMAU addresses found above and the index-select operator are attached to the associated selection operation.

In the third phase of the query translation and optimization process, each operation in the optimized query is replaced by its equivalent set of

PCS primitives. When this phase is completed, the query is placed on the ready list, maintained by the MBC, waiting for execution.

(c) *Scheduling and Monitoring the Query Execution*

The MBC schedules and monitors the execution of the different parts of the user query on the PCs. It synchronizes their processing and signals their termination. The MBC also supervises the data movements between the MSS and the PCS.

One final note regarding the MBC. The implementation of the latter subsystem is strongly dependent on the way the earlier stated functions are partitioned between the front-end computer system and the MBC. Based on this partition, the MBC can be implemented using a powerful mini/micro computer.

3.2.2. The Mass Storage Subsystem

The mass storage subsystem (MSS), shown in Figure 3.2, is the repository of the database and its index-term index. This index is needed because the MSS subsystem is not fully associative.

The MSS is designed as a two-level memory system, as shown in figure 3.2, the mass memory (MM) and the parallel buffer (PB) levels. While the mass memory helps MSS to meet the large capacity storage requirement, the PB helps it to take advantage of the local and sequential references to the database. In the following paragraphs, both of these levels are outlined.

The Mass Memory

The mass memory is organized as a set of moving-head disks, controlled and managed by the mass storage controller (MSC). Each disk is provided with the capability of reading/writing from/to more than one track in

parallel. Tracks which can be read/written in parallel from one disk form what is called the minimum access unit (MACU).* The tuples, within the latter unit, are laid out on the moving-head disk's tracks in a "bit serial-word serial" fashion. The MACU is the smallest accessible unit of data as well as the unit of data transfer between the MM, the PB and the PCS. The MACU in the new machine stores only one MAU. Although the number of tracks in an MACU is limited by the hardware cost and the processing clusters capability, the MACU is expected to have the size of a moving-head-disk cylinder.

In addition to the database relations, the MM stores another type of data, namely, the index terms. In general, the index terms which are defined on attributes of different relations can reside in the same IMAU. In order to improve their retrieval cost, the index terms are clustered together according to their relation and attribute names. That is, the index terms which are defined on the same relation and attribute are likely to reside in the same IMAU.

The IMAU is stored on one MACU. Figure 3.3 shows the layout of an IMAU on the tracks of a moving-head disk. Every track within the latter unit contains a set of blocks of suitable size (~ 4 Kbytes). Each block contains index terms defined for the same relation and attribute. For storage as well as processing efficiency, the < relation-name, attribute-name > common to all the index terms of the block is stored only once (at the beginning of the block). The rest of the block stores only the < value, MAUA > part of the corresponding index terms.

The Parallel Buffer

*Actually, the number of tracks within one MACU can only be one of those in the set { n, n/2, n/4,..., 1} where n is the number of tracks in one cylinder of the moving-head disk and even.

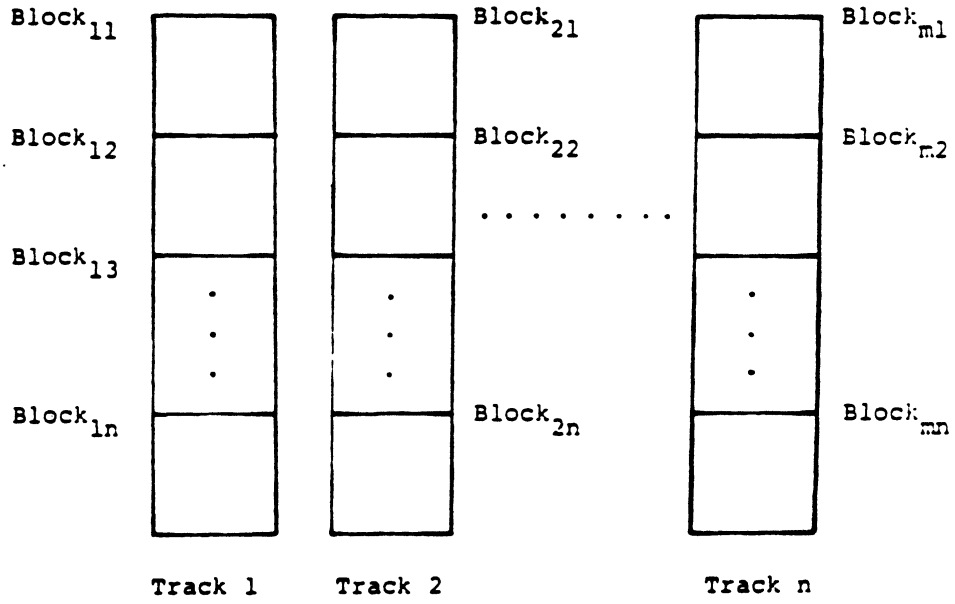

Figure 3.3 The IMAU layout on the physical storage

The primary objective of the parallel buffer (PB) is to help the MSS to take advantage of the fact that most references to the database and index terms are of sequential and local nature. Thus by storing the most frequently referenced data most of the time in the relatively fast PB rather than in the relatively slow MM, the MSS response time (bandwidth) is improved substantially.

The parallel buffer (Figure 3.2) is organized as a set of blocks, each with size equal to that of an MACU. A block is further partitioned into a set of subblocks. Each subblock can buffer one track of a moving-head disk. The parallel buffer is managed by the mass memory controller. The parallel buffer implementation can take advantage of the promising magnetic bubble memory [CHEN78, COMP79] and the charge-coupled device memory[TZOU80] technologies. Both technologies currently have off-the-shelf memory chips which can buffer an entire disk track.

3.2.3. The Processing Clusters Subsystem

The processing clusters subsystem (PCS) is organized as a multiple single instruction stream- multiple data stream (MSIMD) system. The PCS (Figure 3.2) consists of a set of processing clusters which share a common buffer, the parallel buffer. A processing cluster(PC), shown in Figure 3.4, has a single instruction stream-multiple data stream (SIMD) organization. A PC consists of a set of triplets, each of the form:

< I/O controller (IOC), triplet processor (TP), local memory unit (LMU) >.

The set of triplets within a PC is controlled and managed by the cluster master processor (CMP). This processor accesses its triplets through a broadcast bus, the master bus (MBUS). The MBUS permits the CMP to write the same data to all the LMUs of its cluster triplets, simultaneously. On the other hand, the MBUS permits the CMP to sequentially read data from one of its triplets'

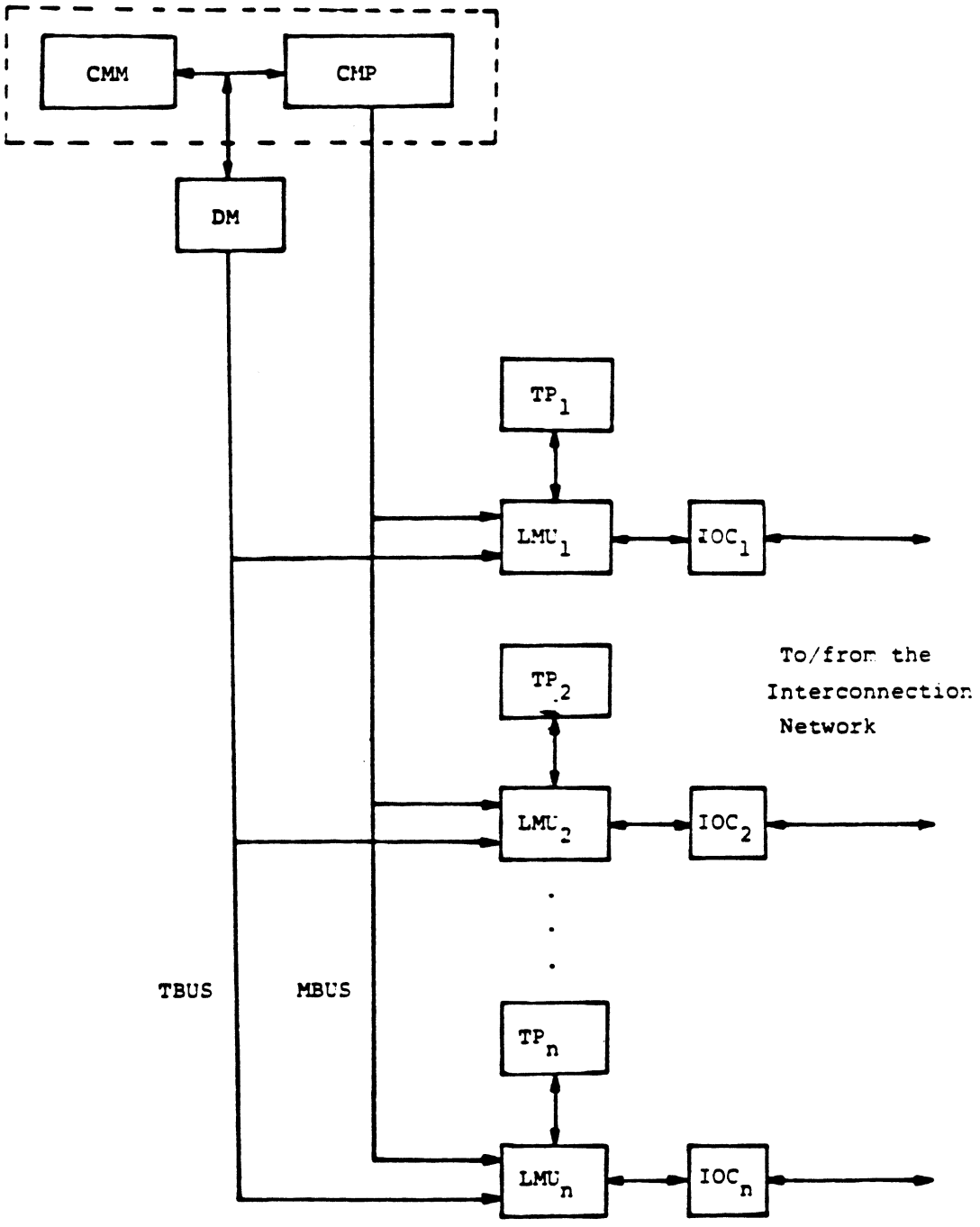

Figure 3.4 The Processing Cluster Organization

LMUs.

Within a PC (Figure 3.4), the data is moved between its triplets via a bus, the triplets bus (TBUS), controlled by a high-speed DMA controller, the data mover (DM). Under instructions from the CMP, the DM moves data items between the LMUs of the cluster's triplets. The TBUS is provided with both point-to-point as well as broadcast capabilities.

In general, the LMU i in a PC is accessible directly by the CMP; through the MBUS, the data mover through the TBUS, and both the i^{th} triplet processor and the i^{th} I/O controller. Although a LMU is expected to have relatively large capacity (multiple the size of a moving-head-disk track), nevertheless, it will be implemented using random access memory (RAM) storage technology. RAM has the ability to support the processing of the relational database operations, in a triplet, in a more cost-effective way than other technologies, such as the charge-coupled devices memory technology, especially in the wake of the tremendous improvements in the per bit cost of RAM and the near future availability of the 256-Kbit chips as off-the-shelf units. An IOC and a TP of a triplet are expected to be implemented as a high-speed DMA controller and an off-the-shelf microprocessor, respectively.

3.2.4. The Interconnection Network Subsystem

The Interconnection Network is designed to fulfill the following three important requirements:

1. The ability to interconnect the PCs, the MM moving-head disks and the PB blocks.
2. The ability to support the parallel processing of the system's most important operations (selection, projection, ϑ-join and index-select). Also, the network must facilitate the simultaneous processing of dif-

ferent system operations. This implies that the network must enable the PCs to individually read/write from/to different blocks of the PB, simultaneously as well as allowing more than one PC to simultaneously read from the same block of the PB. In other words, the network should possess the crossbar interconnections (between the PCs, the PB blocks and the MM moving-head-disk units) as well as broadcasting capabilities (between the PB blocks and the PCs).

3. The network simplicity. That is, the network must be simple enough to be able to support a relatively large number of PCs, PB blocks and MM moving-head disks.

The interconnection network subsystem (INS), shown in Figure 3.2, is a modified version of an interconnection network proposed by Dewitt[DEWI79]. Dewitt proposed a network which is a modified version of the crossbar switching network. Adopting Dewitt's network to our proposed machine would result in the interconnection of every triplet of a PC to every subblock of the PB blocks. In Dewitt's network the role of the buffer and the processors are interchanged. Traditionally, in a crossbar switching network, the processors play the active part while the buffer subblocks play the passive ones. In Dewitt's network, the active part is played by the buffer subblocks where each one continuously broadcasts its contents along a 1-bit-wide bus. Whenever a processor wants to read a page, it switches itself to the specific subblock bus. Thus more than one processor can simultaneously read the same subblock.

Although Dewitt's network provided a neat solution to some of the interconnection system requirements, it still has several drawbacks. The most important one is its inability to support a large number of triplets and subblocks. The logic complexity at the triplet/disk-head grows as the number of

PB subblocks is multiplied by the number of triplets/disk-heads in the system [QADA80]. Thus for a large number of triplets, MM moving-head disks and PB subblocks, Dewitt's network becomes prohibitively expensive.

The INS is a modified version of Dewitt's network. In this network only one triplet/track in each PC/(MM moving-head disk) is connected to the same PB subblock (via the corresponding 1-bit-wide bus). Whenever a PC(s)/(MM disk) needs to read from a given PB block, its IOCs/(disk-track heads) need only to switch themselves to the appropriate set of data buses. If the parallel buffer block contains a data MAU then the IOCs/(disk heads) proceed to read it starting at a tuple boundary. However, for an index MAU the IOCs/(disk heads) proceed to read it starting at an index block boundary.

Whenever a PC/(MM disk) needs to write to a given Parallel Buffer block, its IOCs/(disk heads) need only to switch themselves to the appropriate set of data buses. The writing then follows immediately. Notice that the MSC is responsible for preventing any two PCs or disks, or both, to write to the same parallel buffer block.

Figures 3.5 and 3.6 show two examples of the proposed network. The network in Figure 3.5 interconnects one processing cluster, one moving-head disk and one PB block. The network in Figure 3.6 interconnects two processing clusters, each having three triplets, one moving-head disk and two PB blocks, each with three subblocks. Notice that (each triplet within a PC)/(each head of a disk) is connected to one subblock in each block of the PB.

The newly proposed interconnection network has a logic complexity, at a triplet or disk head, smaller than that of Dewitt's network by a factor equal to the number of subblocks within one block (NS). Thus the new network has the ability to support a large number of triplets and MM moving-head disks.

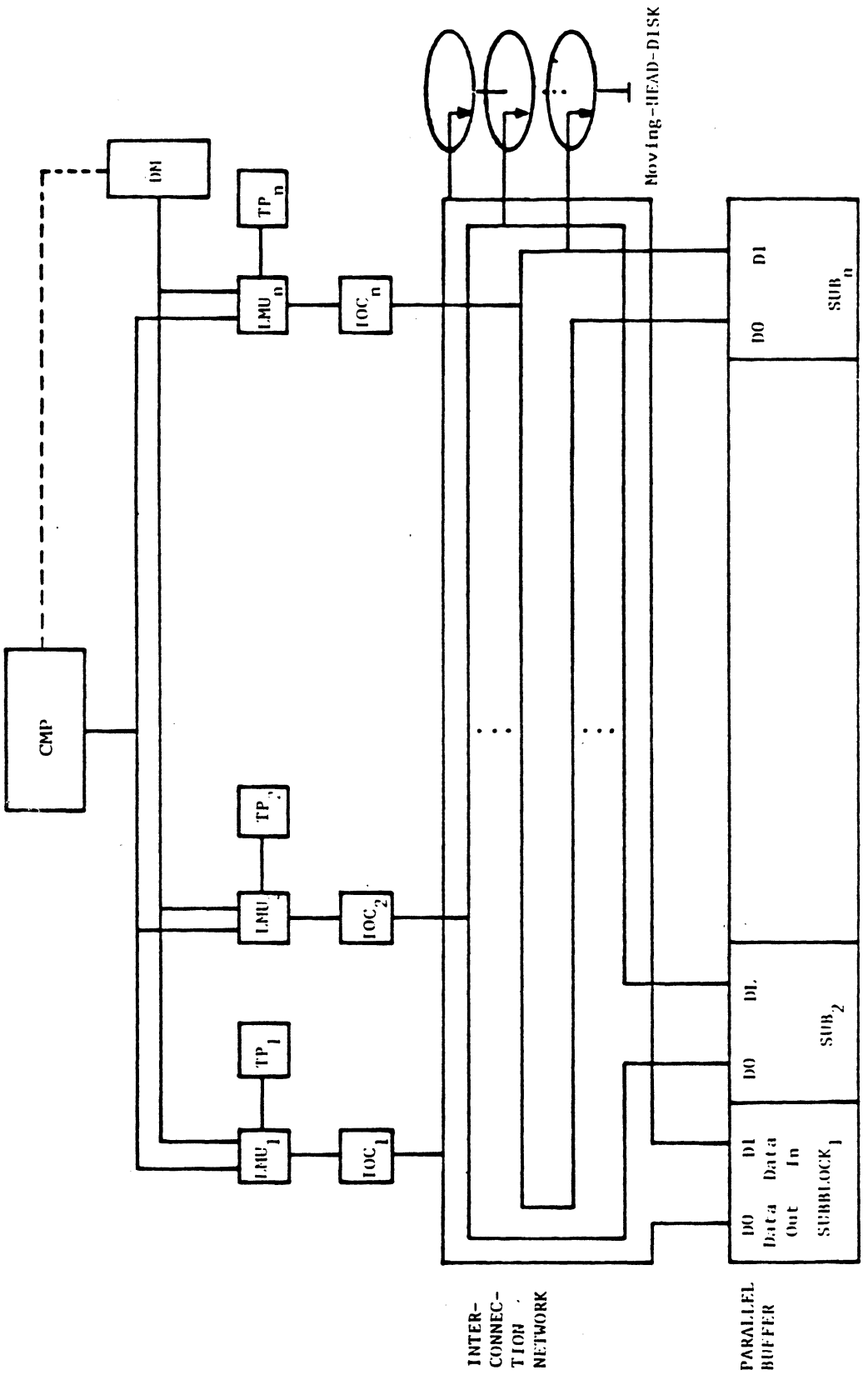

Figure 3.5 An Example of the Interconnection Network with One Processing Cluster.

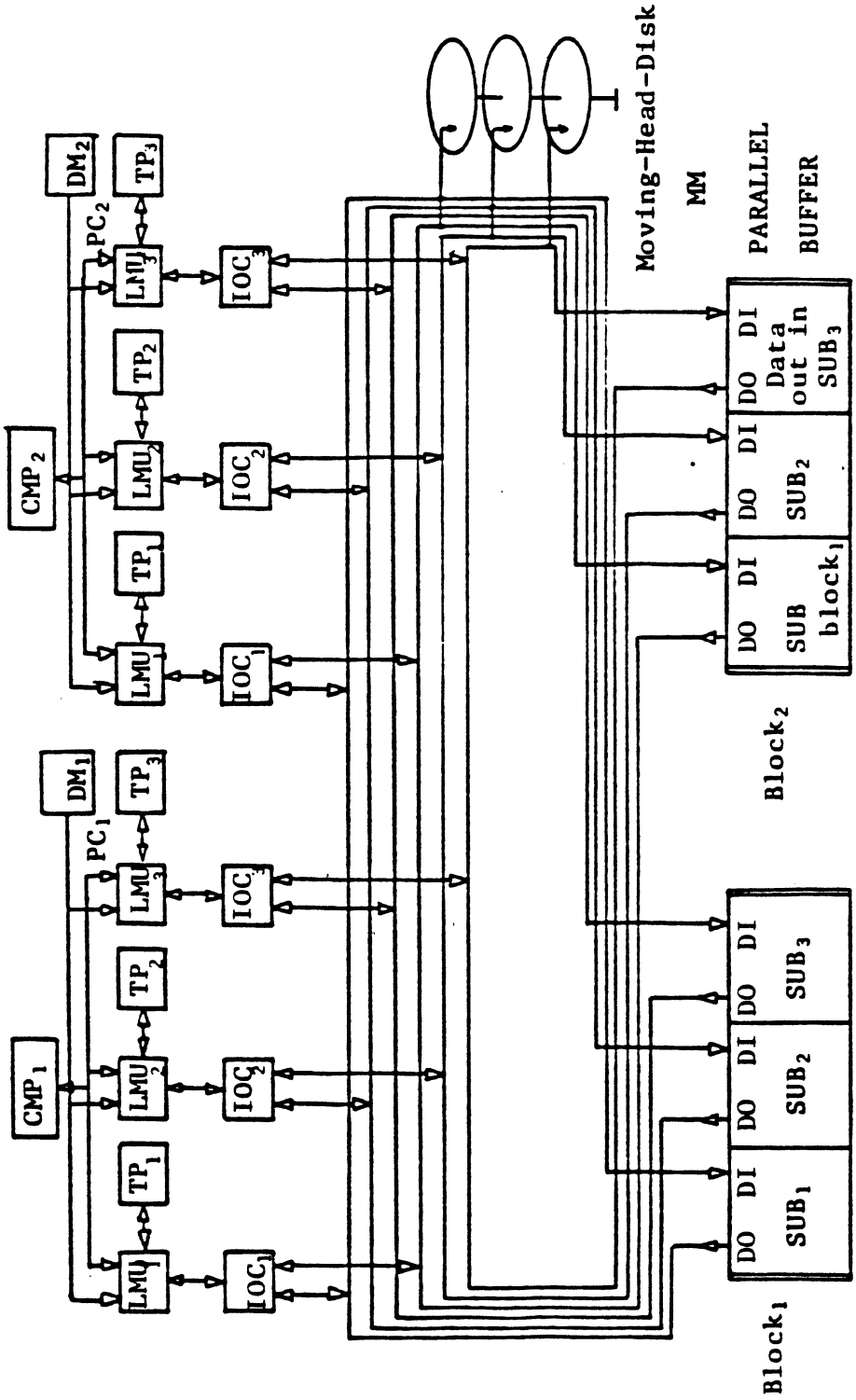

Figure 3.6 An Example of the Interconnection Network with two Processing Clusters.

That is, for almost the same hardware cost, the new network is able to support NS times the number of triplets and MM disks that can be supported by Dewitt's network.

CHAPTER 4

THE RELATIONAL θ-JOIN

The ϑ–join is one of the most important operations of the relational data model. It participates in all data retrieval queries which reference more than one relation. The ϑ–join* operation takes two relations, the "source (S)" and "target (T)" relations, together with a predicate P to produce the "output (O)" relation. Based on the type of the operator "ϑ" associated with the predicate p, the ϑ–join operation has different types. These types can be grouped into two categories, namely, the equi-join and the nonequi-Join ones. An equi-join operation is a ϑ–join with ϑ being the operator"=." On the other hand a nonequi-join operation is a ϑ–join with $\vartheta \varepsilon \{\neq, >, \geq, <, \leq\}$.

In the proposed RDBM one or more PC is used to perform the ϑ–join operation. In general, the number of PCs assigned to perform such an operation is an MBC decision. This decision is based on many factors, such as the size of the input relations, the number of available PCs and the priority class to which the operation query belongs.

The flexibility and generality of RDBM architecture permits the implementation of a large set of algorithms for the equi-join operation. A subset of that can also perform the nonequi-Join operations. In all of these algorithms, the MBC starts the execution of a ϑ–join operation by broadcasting the operation code and the tuple format of both the source and target relations to the set of PCs which participate in the execution of the operation. The cluster master pro-

*For the formal definition of the ϑ–join refer to Section 1.3

cessors, in turn, broadcast this information to their triplets.

In Section 4.1, the set of algorithms which perform the equi-join operation on the RDBM is presented. This presentation will not pay much attention to the detailed implementations of these algorithms. In Section 4.2, analytical models for the execution of the presented algorithms on the proposed RDBM are developed. In section 4.3, these models are used to evaluate the performance of the equi-join algorithms. This evaluation is used in choosing the best performing algorithm(s) for the different input data environments. Finally, in Section 4.4, the models of Section 4.2 are used, with slight modifications, to evaluate the effect of improving the PC intertriplets communication on the performance of the equi-join algorithms.

4.1. Algorithms for the Equi-Join Operation

The flexibility and generality of RDBM architecture permits the implementation of a large set of algorithms for the equi-join operation. These algorithms can be grouped into four categories, namely, the basic Equi-Join algorithms, the "target relation partial filtering (TPF)" equi-join algorithms, the "source-target relations partial filtering (STPF)" equi-join algorithms and the "source-target relations complete filtering (STCF)" equi-join algorithms. In the following subsections the algorithms within each of these categories are presented.

4.1.1. The Basic Equi-Join Algorithms

The basic algorithms category is comprised of twelve different algorithms. In Section 4.1.1.1, a scheme to classify and name these algorithms is introduced. The scheme rests on several attributes which characterize the basic algorithms. These attributes are also introduced and discussed. In section 4.1.1.2, the sequence of steps a typical PC goes through while executing a

basic algorithm is presented. In Section 4.1.1.3, the memory space (within a PC) required to support the basic algorithms is presented.

4.1.1.1. The Basic Equi-Join Algorithms Classification Scheme

The basic equi-join algorithms can be classified according to three common attributes. The first attribute is the way a particular algorithm distributes the tuples of the source and the target relations among the PCs for processing. The second attribute is the way a particular algorithm distributes a PC's share of the source and the target tuples among the PC's triplets for joining. The third attribute is the way a particular algorithm performs the equi-join operation within a PC triplet. The classification of the basic algorithms according to these three attributes is presented in Figure 4.1.

Two methods exist for distributing the tuples of the source and target relations among the PCs for processing. These two methods are:

(a) The Global Broadcast Method

In this method, each PC ε APC* is assigned a different MAU of the source relation. Then every MAU of the target relation is broadcast, one MAU at a time, to all PC ε APC. The latter joins the tuples of the source MAU with all the tuples of the target MAUs. This process is repeated until all the MAUs of the source relation have been processed.

(b) The Global Hash Method

In this method, a hashing function partitions the join attribute underlying domain into disjoint subsets (global buckets) such that the expected number of tuples from the source relation per subset would fit in the PC's local store. Then each PC ε APC is assigned a different global bucket to join.

*APC is the set of processing clusters assigned to execute the equi-join operation.

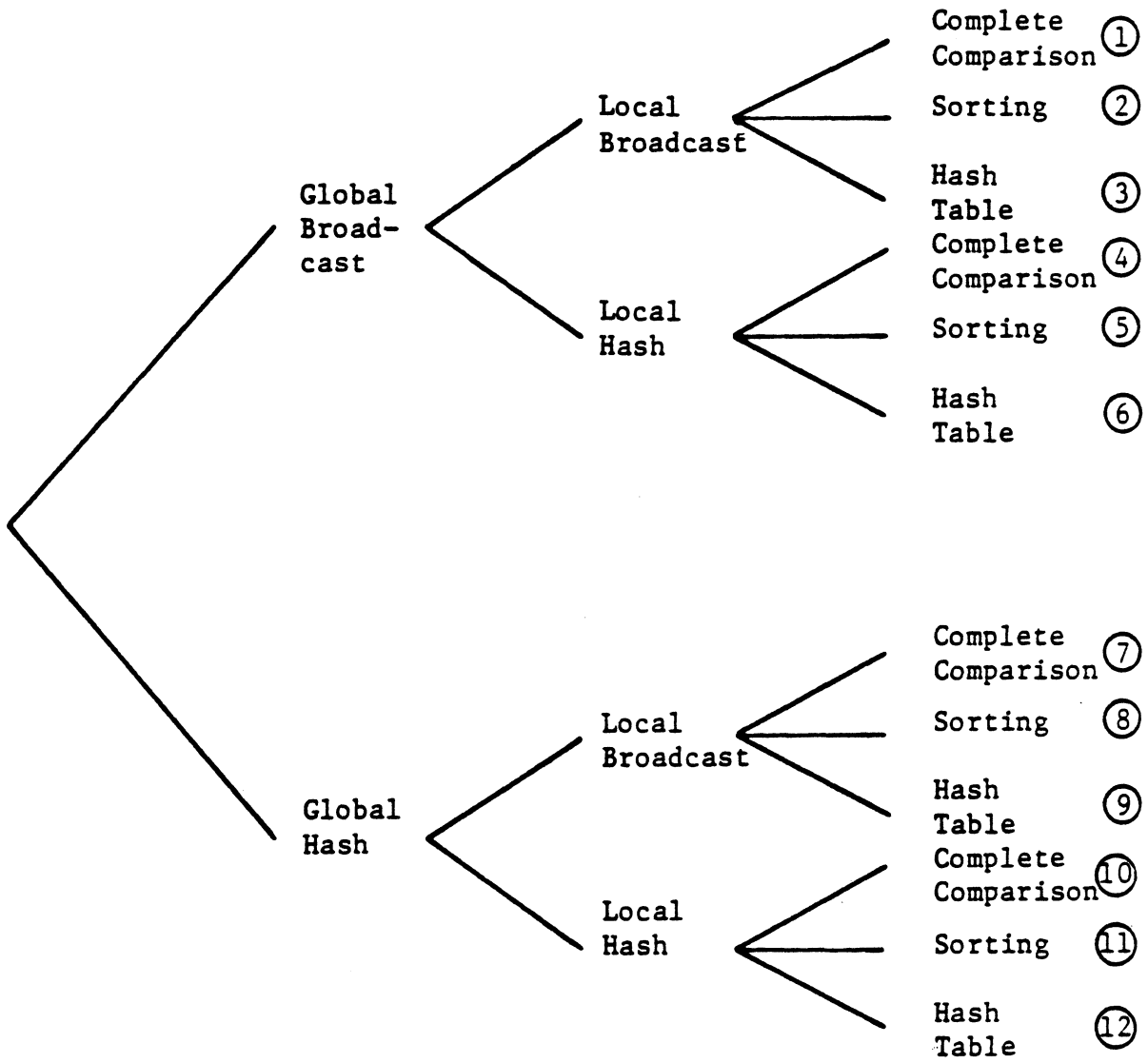

Figure 4.1 The Basic Equi-Join Algorithms

The latter step is repeated until all global buckets have been processed.

A PC ε APC collects the tuples of the source and target relations which fall in a subset i as follows:

> The PC reads an MAU of the source relation into its local memory units. Then the tuples in an LMU are processed by the corresponding triplet processor. A typical tuple is processed by computing a hashing function. The input to the hashing function is the tuple's join attribute value. The output of the hashing function is a subset number j to which the tuple would belong. If $j=i$, then the tuple will be kept in the LMU for further processing, otherwise it will be deleted. This process is repeated for all the MAUs of the source relation. Then it is repeated for all MAUs of the target relation.

From the preceding, it is concluded that the execution of a basic equi-join algorithm is composed of a number of phases. During a typical phase, one PC ε APC is assigned some source and target tuples for joining. If the equi-join operation is performed using the global broadcast method, then the PC is assigned one MAU of the source relation and all the MAUs of the target relation. The number of phases, in this case, is equal to that of the source MAUs. On the other hand, if the equi-join operation is performed using the global hash method, then, during one phase a PC is assigned the tuples of a global bucket. The number of phases, in this case, is equal to that of the global buckets.

The tuples assigned to a particular PC during the execution of one phase of the equi-join operation can be distributed among the PC's triplets using one of the following two methods:

(a) The Local Broadcast Method

In this method, the tuples of the source relation assigned to the cluster triplets are not redistributed. The assigned tuples of the target relation are broadcast by the data mover, one tuple at a time, to all the triplets within the PC. Whenever a target tuple is broadcast to a particular triplet, the corresponding processor joins it with the triplet's share of the source tuples.

(b) The Local Hash Method

In this method, the tuples of the source relation assigned to a PC during one phase of the equi-join execution are hashed, based on the tuples' join attribute values, to the PC triplets. A triplet stores, in its LMU, these tuples of the source relation which hash to itself.

The assigned tuples of the target relation are also hashed, based on the tuples' join attribute value, to the PC's triplets. Whenever a target tuple hashes to a particular triplet, the corresponding triplet processor will join it with the triplet's share of the source tuples.

Three methods are available, at the disposal of a triplet, for joining one tuple of the target relation with the tuples of the source relation assigned to it during the execution of one phase of the equi-join operation. These methods are:

(a) Complete Comparison Method

In this method, the source tuples assigned to a triplet are stored in its LMU in a random fashion. A target tuple is joined with the source tuples as follows:

> Scan the source tuples one tuple at a time. For every source tuple, compare its join attribute value with that of the target tuple. If they match, then concatenate both tuples and move the result tuple to the output buffer.

(b) Sorting the Source Tuples Method

In this method, the source tuples assigned to a triplet are stored, in its LMU, in a sorted order. The source tuples are sorted on their join attribute values. A target tuple is joined with the source tuples as follows:

> Using the binary search method[KNUT73], locate those source tuples whose join attribute values are the same as those of the target tuple. Concatenate these tuples with the target one and move the result tuples to the output buffer.

In a triplet, the source tuples sorting step will not be implemented by actually sorting the source tuples but by sorting a table of addresses through which the source tuples can be referenced. The storage allocated for the source tuples within a triplet is divided into two areas, namely, the primary and the secondary ones. In the secondary area, the source tuples are stored in linked lists structure. Each linked list corresponds to one join attribute value. The headers of the linked lists are stored, in the primary area, in a sorted way. This implementation avoids the high cost of the tuples' movement which could result from the actual sorting of the source tuples.

(c) Hash Table Method

In this method, the source tuples assigned to a triplet are stored, based on their join attribute values, in a hash table. A target tuple is joined with the source tuples as follows:

> **Hash the target tuple, based on its join attribute value, to one of the hash table buckets. For every source tuple within the bucket compare its join attribute value with that of the target tuple. If they match, then concatenate both tuples and store the results in the output buffer.**

The hash table of a triplet is implemented as two storage area's, namely, the primary and the secondary ones. In the secondary area, the source tuples which hash to the same bucket are stored in a linked list structure. The headers of the linked lists are stored in the primary area. This implementation improves the triplet's LMU storage utilization.

The different basic equi-join algorithms can be named using the preceding scheme. An algorithm name can be thought of as the ordered quadruple $<nam_1, nam_2, nam_3, basic>$, where nam_1 is the name of the method by which the algorithm distributes the tuples of the source and target relations among the participating PCs for processing; nam_2 is the name of the method by which the algorithm distributes the tuples of the source and target rela-

tions, assigned to a PC during one phase, among its triplets for processing; nam_3 is the name of the method by which the algorithm performs the equi-join operation within a triplet and the word "basic" indicates that the algorithm belongs to the basic group. For example, algorithm number 1 of Figure 4.1 is called the "global broadcast-local broadcast-complete comparison-basic" algorithm. For simplicity this name will be abbreviated to BBC-Basic. In the same way, algorithm Number 8 of figure 4.1 is called "global hash-local broadcast-hash table-basic (HBH-Basic)" algorithm.

4.1.1.2. Executing the Basic Algorithms by a Processing Cluster"

In general, the processing of a basic algorithm can be decomposed to a number of phases. Throughout the execution of the basic algorithm, a typical PC executes one or more of the corresponding phases. A typical phase, as shown in Figure 4.2, is divided into six subphases. During the first subphase, a set of the source relation tuples (the tuples of a source MAU for the global broadcast algorithms, the tuples of the source relation which hash to a global bucket for the global hash algorithms) are selected by the PC triplets. During the second subphase the CMP in cooperation with the cluster triplets and the data mover (do nothing with)/(hash to the cluster triplets) those tuples of the source relation which were selected during the first subphase. During the third subphase every triplet, within the PC, (do nothing with)/sort/(store in a hash table) its share of the source tuples. In RDBM a PC executes these subphases in a pipeline fashion with the tuple as the unit of the pipeline. That is, as soon as a tuple is selected in the first subphase it will trigger the execution of the second subphase which in turn will trigger the execution of the third one. When all the tuples of the source relation, which were selected in the first phase, are processed through the second and third ones, then the fourth subphase will start to be executed.

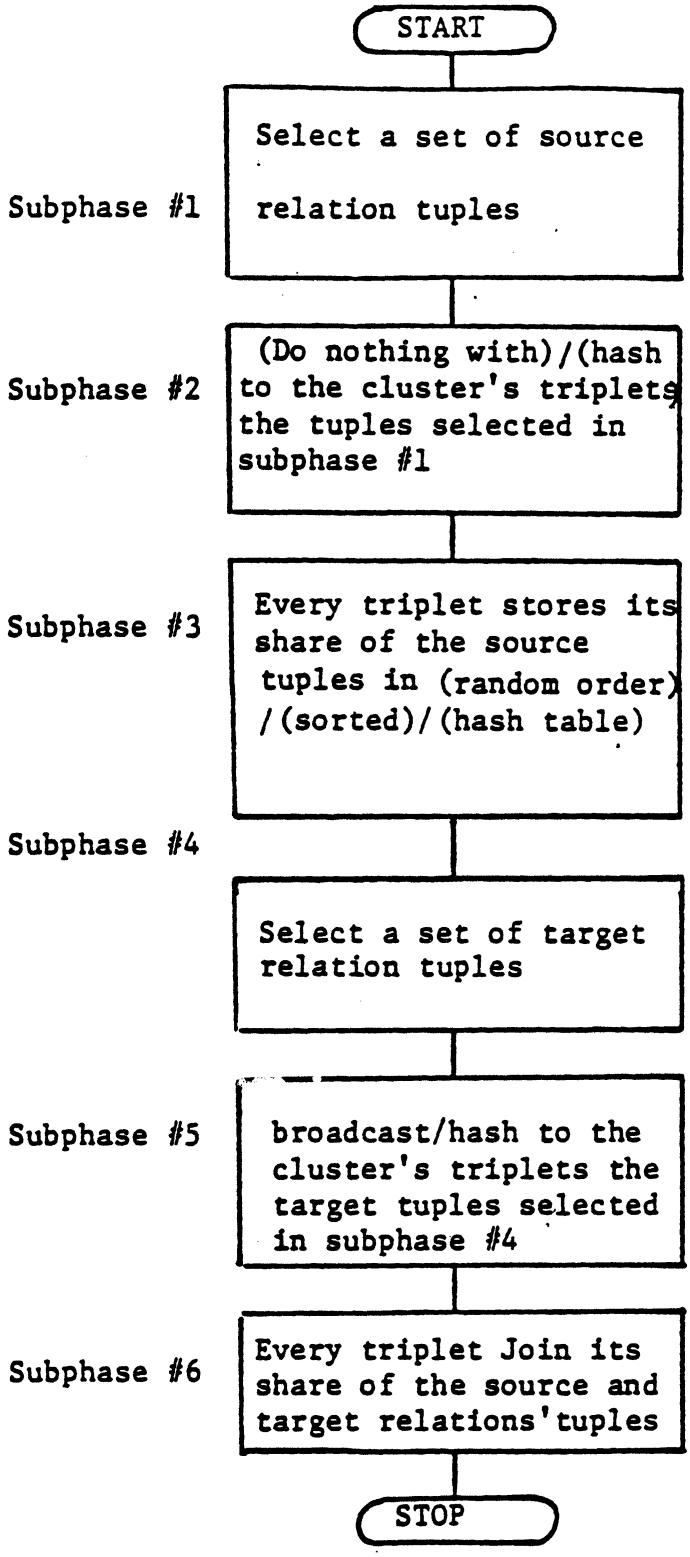

Figure 4.2 A Typical Phase in Executing A Basic Equi-Join Algorithm

During the fourth subphase a set of the target relation tuples (all the target relation tuples for the global broadcast algorithms, the tuples of the target relation which hash to a global bucket for the global hash algorithms) are selected by the PC triplets. During the fifth subphase the cluster master processor, in cooperation with the cluster triplets and the data mover broadcast/(hash, based on the join attribute values) to the cluster triplets those tuples of the target relation which were selected during the fourth subphase. During the sixth subphase every triplet, within the PC, joins its share of the source and target tuples. Just as for the first, the second and the third subphases, the PC executes the fourth, the fifth and the sixth subphases in a pipeline fashion. That is as soon as a target tuple is selected in the fourth phase, it will trigger the execution of the fifth phase (broadcast/hash the target tuple), which in turn will trigger the execution of the six phase [joining the target tuple with the source tuples in the triplet(s)].

4.1.1.3. The Basic Equi-Join Algorithms Memory Requirement

In order to support the basic equi-join algorithms, the local memory unit of a typical triplet must, at least, have three buffers. The first buffer stores the triplet's share of the source tuples. This buffer is called the source buffer (BUFS). The second buffer stores those tuples (source/target) being read from the parallel buffer. This buffer is called the input buffer (BUFI). The third buffer stores those tuples of the output relation waiting to be written to the parallel buffer. This buffer is called the output buffer (BUFO). BUFI and BUFO must each have the capacity of the track of a moving-head disk (a parallel buffer subblock). An additional buffer with capacity equal to that of BUFI is needed if overlapping the processing of the PC's triplets and the read/write from/to the parallel buffer is required. Some small extra memory space is needed, in a local memory unit, to

s upport the algorithms' data structure and the communication between the m aster cluster processor and its triplets.

To support the "global broadcast" basic algorithms, BUFS must have e nough capacity to store one or more tracks of the source relation. To sup- p ort the " global hash" basic algorithms, BUFS must have enough capacity to s tore those tuples of the source relation which hash to the same global b ucket.

4.1 .2. The TPF Equi-Join Algorithms

The algorithmic category TPF is comprised of twelve different algo- rit hms. Every algorithm within this category is an extension of one of the ba sic equi-join algorithms. As for the basic algorithms, a TPF algorithm is de composed into a number of phases, each with six subphases. During the se cond, the third, the fifth and the sixth subphases, the same processing as th at carried out for the corresponding subphases of the basic algorithms are pe rformed. During the first subphase, in addition to selecting a subset of the so urce relation tuples, the TPs of the PC encode the join attribute values of th ese selected tuples. The encoding scheme involves a hashing function and a ve ctor of suitable number of components (referred to as the (cluster) vec- to r). Each component has the size of 1 bit. This vector is initialized and main- ta ined by the CMP. The hashing function transforms the join attribute value of a selected tuple to an index to the vector. This index is passed by the TP to th e CMP. The CMP "sets" the corresponding vector bit to one. During the fo urth subphase and before further processing of a selected target tuple, its joi in attribute value is checked against the encoded set of the source tuples' joi in attribute values. This is done by transforming, using the hashing func- tio n of the first subphase, the target tuple's join attribute value into an index to a bit within the vector. If the corresponding bit is set then the target tuple

will be hashed/broadcasted to one/all triplet(s) for further processing. If the bit is not set then a source tuple with join attribute value which matches that of the target one does not exist, in the PC's local store, and no further processing for the target tuple is needed.

Since every algorithm in the TPF category is an extension of one of the basic algorithms, the same scheme, which is used to classify the latter algorithms, is used to classifying the TPF algorithms. Also the same naming convention as that adopted for the basic algorithms is used for the TPF algorithms. However in the TPF, the word basic in the name of a basic algorithm must be replaced by the word TPF. For example, algorithm Number 8 of Figure 4.1 has the name "global hash-local broadcast-hash table-TPF." For simplicity this name is abbreviated as "HBH-TPF."

In order to support the TPF algorithms, the local memory units of a PC must have the same capacity as those required to support the basic ones. However, in addition, a TPF algorithm requires some additional memory space, in the CMP memory, to support the vector.

4.1.3. The STPF Equi-Join Algorithms

The algorithmic category STPF is comprised of six different algorithms. Every one of the latter algorithms is an extension of one of the "global hash" TPF algorithms.* As in the TPF algorithms, a PC executing one of the STPF algorithms goes through a number of phases, each with six subphases. During the first subphase the PC's triplets collect the tuples of the target relation which hash to the global bucket corresponding to the PC . In addition to

---

Actually, the algorithmic category STPF is comprised of twelve different algorithms, each is an extension of one of the TPF equi-join algorithms. However, the STPF algorithms which correspond to the "global broadcast" TPF equi-join algorithms are not considered. This is because each of these algorithms will require any PC to have enough space, in its LMUs, to store all the tuples of the target relation or to read the tuples of the target relation off the disk twice during the execution of one equi-join phase. Thus resulting in an inferior algorithm relative to the others proposed. On the other hand, this is not the case with the "global hash" TPF algorithms.

that, the TPs of the PC encode the join attribute values of the collected tuples. The encoding is done using a hashing function, computed by the PC's triplets and a vector HBIT-T, initialized and maintained by the CMP. When all the collected target tuples have been processed through the first subphase, execution of the second subphase will start.

During the second subphase, the PC's triplets collect the tuples of the source relation which hash to the current global bucket. In addition, the join attribute value of every collected tuple is encoded using a vector HBIT-S and the hashing function of the first phase. In addition, the collected tuples are checked against the encoded set of the target tuples' join attribute values. A bit, in HBIT-S, set due to a source tuple is compared with the corresponding bit of HBIT-T. If the corresponding bit in HBIT-T is set then the source tuple is retained for further processing, otherwise it is discarded.

During the third subphase, the source relation tuples which survive the checking process of the second subphase are (left in)/(hashed to the PC's triplets). During the fourth subphase, the source tuples within a triplet are stored in a random/sorted/hash table form. The PC executes the second, the third and the fourth subphases in a pipeline fashion. That is as soon as a source tuple is selected it is checked against HBIT-T. If the tuple survives the checking then it will be stored in the proper triplet in a random order/sorted order/hash table. The execution of the second, the third and the fourth subphases continues until all the tuples of the source relation, selected by the PC, have been processed. This triggers the execution of the fifth subphase.

During the fifth subphase, the join attribute value of every target tuple, selected during the first subphase, is checked against the encoded set of the source tuples' join attribute values. If the corresponding bit, in HBIT-S, is not set then the target tuple is discarded. If the bit is set then the tuple is

(broadcasted to)/ (hashed to one of) the cluster's triplets for further processing. During the sixth subphase every target tuple which survives the checking process is joined with the source tuples stored within the PC's triplet(s). Notice that the PC executes the fifth and sixth subphases in a pipeline fashion.

Since every algorithm in the category STPF is an extension of one of the "global hash" TPF algorithms, the same classification scheme which was presented for the latter can be used for classifying the STPF algorithms. Also, the same naming convention can be used to name the STPF algorithms. However, in the latter case, the word "TPF" is replaced by the word STPF. For example, algorithm Number 2 of Figure 4.3 has the name "global hash-local broadcast-hash table-STPF." This name is abbreviated as "HBH-STPF."

In order to support the STPF algorithms, the local memory units must have the same capacity as those required to support the basic algorithms (the target relation tuples selected during the first subphase can be stored, waiting for the end of the fourth subphase, in the BUFO and therefore no additional storage space is needed in the local memory units). In addition the STPF algorithms require some additional space, in the memory of the CMP, to support the vectors HBIT-S and HBIT-T.

4.1.4. The STCF Equi-Join Algorithms

The algorithmic category STCF is comprised of twelve different algorithms. Every algorithm within this category is an extension of one of the basic equi-join algorithms. An STCF algorithm is comprised of two phases (Figure 4.4), namely, the global filtering phase and the join phase. The objective of the global filtering phase is to filter out many unmatching tuples of the source and target relation. This phase is followed by the join phase where one of the basic equi-join algorithms is used to join the remaining tuples of the

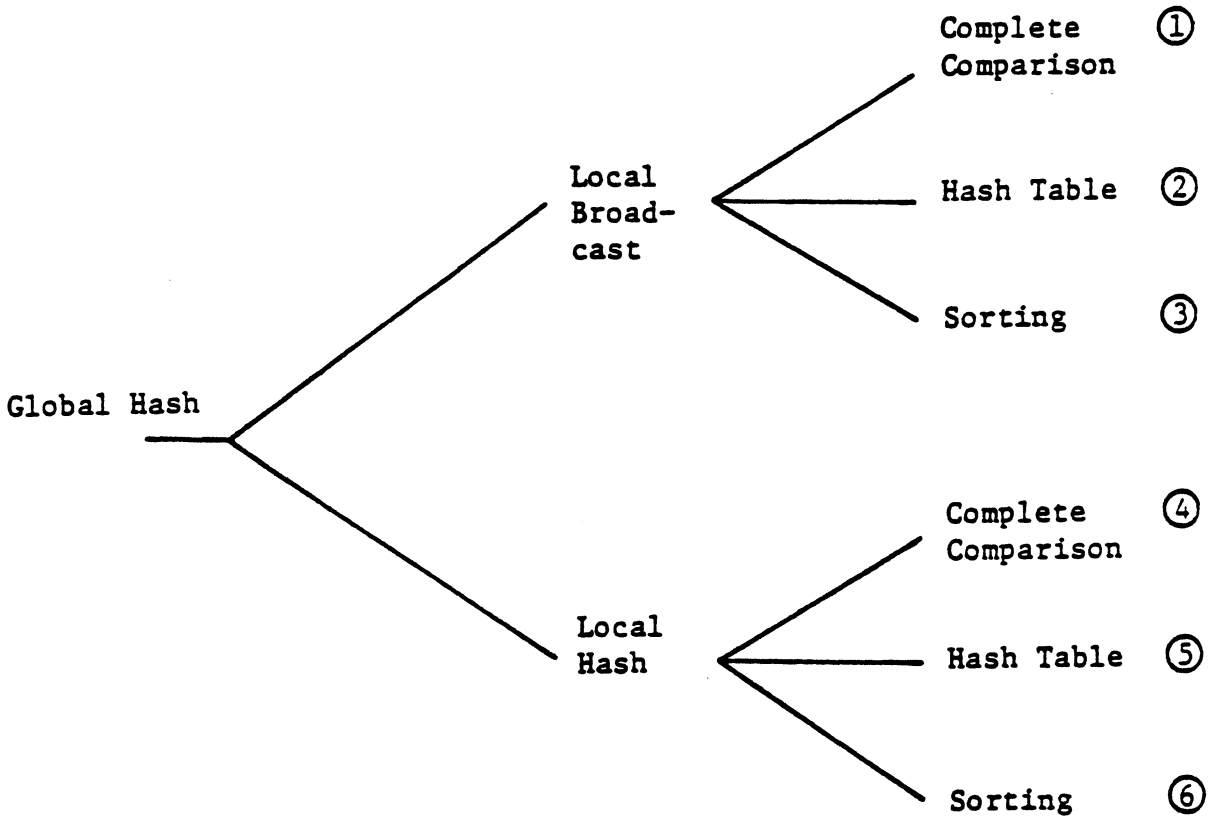

Figure 4.3 The STPF Equi-Join Algorithms

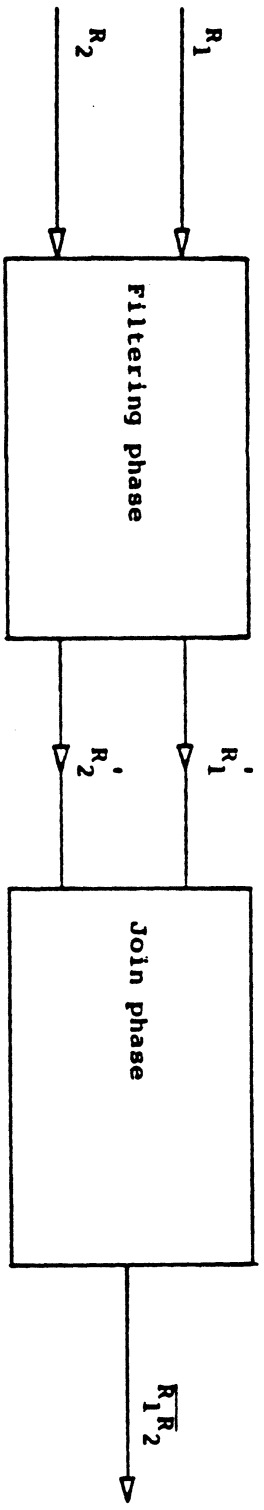

Figure 4.4 The STCF Equi-Join Algorithms

source and target relations.

The global filtering phase is carried out by one PC. This phase can be divided into three subphases. During the first subphase the PC reads all the MAUs, one MAU at a time, which store the source relation tuples. The PC also encodes the join attribute values of the source relation tuples. The encoding is carried out using a hashing function and a vector, HBIT-S, initialized and maintained by the CMP. The hashing function transforms a typical join attribute value into an index to the vector HBIT-S. The CMP "sets" the corresponding vector bit to one.

During the second phase, the PC reads all the MAUs, one MAU at a time, which store the target relation tuples. The PC encodes the join attribute values of the target relation. The encoding is done using the hashing function of the first subphase and a vector (HBIT-T) initialized and maintained by the CMP. The PC also checks the target tuples' join attribute values against the encoded set of the source join attribute values. Every tuple of the target relation which survives the latter checking is moved to the output buffer. Whenever the PC's output buffer fills up, it is written into a block of the parallel buffer. If the parallel buffer has no empty space then some of the parallel blocks are transferred to the moving-head disks before the PC can write its own output buffer.

During the third subphase the PC reads again all the MAUs, one MAU at a time, which store the source relation tuples. Every tuple within an MAU is checked against the encoded set of the target tuples' join attribute values. The tuples of the source relation which survive the latter checking are moved to the output buffer.

During the join phase, one or more PC can be assigned to execute the physical join of the surviving source and target relations' tuples, using one of

the basic equi-join algorithms.

Since every algorithm in the category STCF is an extension of one of the basic algorithms, the same classification scheme which was presented for the latter category can be used for presenting the STCF algorithms. Also the same naming convention as that adopted for the basic algorithms can be used to name the STCF algorithms. However, in the latter case, an algorithm name needs to include the word "STCF" instead of the word "basic."

In order to support the STCF algorithms, the local memory units must have the same capacity as those required to support the basic algorithms. In addition, the STCF algorithms require additional space, in the memory of the cluster master processor, to store the vectors HBIT-S and HBIT-T.

4.2. Models for Executing the Equi-Join Algorithms on the Proposed RDBM

In this section, a series of analytical, average valued models are introduced. Each one, called the execution model, will model the execution of one of the equi-join algorithms on the proposed RDBM. These models are used in the next two sections to evaluate and compare the performance of the equi-join algorithms. They are also used to evaluate the effectiveness of carrying out some architectural changes in the proposed RDBM.

The parameters which characterize the set of the execution models can be grouped into four categories, namely, the data parameters, the hardware parameters, the hardware-algorithm parameters and the performance parameters. The data parameters, the hardware parameters and the hardware-algorithm parameters which characterize an execution model are called the model input parameters. The execution model performance parameters are called the model output parameters. In the following paragraphs the parameters within each of these categories are presented.

(a) The Data Parameters

The data participating in the equi-join operation consists of two relations, namely, the source relation (S) and the target relation (T). These two relations are modeled using the following parameters:

Let $x \in \{ S , T \}$

Then NTx = the cardinality of (number of tuples in) relation x,

LTx = the tuple length (in bytes) of relation x,

LJ = the join attribute length (in bytes),

ND = the cardinality of the domain underlying the Join attribute.

The data model assumes that a tuple of relation S or T is equally likely to take, for its join attribute, any value from the join attribute underlying domain. This assumption eliminates the need to have other parameters to characterize the data model, such as the number of distinguished values in both the source and target relations' join attribute. With this assumption the value of the latter parameters (as is shown later) can be obtained from the values of the parameters NTS,NTT and ND.

(b) The Hardware Parameters

The RDBM being modeled consists of one PC , one parallel buffer block and one moving-head disk. The PC has been modeled using the following parameters:

NP = the number of triplets per PC,

$BUFSC$ = the capacity of the storage, in the PC's LMUs, allocated for buffering tuples from the source relation,

TCD = time (in ms) to directly compare the join attribute value of a source tuple with that of a target tuple,

TCI = time (in ms) to indirectly compare (access the tuples through a table of pointers) the join attribute value of a source tuple with that of a target one,

TH = time (in ms) to calculate a hashing function with the tuple's join attribute value as the input argument,

TMy = time (in ms) to move a tuple of the relation y within the LMU of a triplet, where $y \varepsilon \{ S, T, O \}$ and O is the output (result) relation,

TEP = time (in ms) to swap two pointers within the LMU of a triplet,

Tx = time (in ms) to move a tuple of relation x across the TBUS, $x \varepsilon \{ S, T \}$.

In the hardware model, it is assumed that the MAU is a cylinder of the moving-head disk. The moving-head disk and the parallel buffer have been modeled using the following parameters:

MAUC = the MAU (moving-head disk cylinder) capacity (in bytes),

TDAC = moving-head-disk average access time (in ms),

TSK = time (in ms) for the moving-head disk to seek one track,

TDT = time (in ms) to transfer an MAU (cylinder) between the moving-head disk and the parallel buffer block,

TBT = time (in ms) to transfer an MAU between the parallel buffer block and the triplets.

(c) The Hardware-Algorithm Parameters

Recall that some of the equi-join algorithms use some data structures such as a hash table and a vector. These two structures have been modeled using the following two parameters:

NBIT = the number of bits in the vector,

NBP = the number of buckets in the hash table of a triplet.

(d) The Model Output Parameters

The comparison of the different equi-join algorithms, executed by RDBM, is done mainly through comparing the behavior of an important performance measure (the model output parameter), namely, the "total execution time (TTIME)." /The TTIME is the time to execute an equi-join algorithm on RDBM without any overlap between the activities of the different hardware units.

Basic Assumptions:

In developing the equi-join execution models, several basic assumptions have been made, namely:

1. A tuple of the relation S or T is equally likely to carry, in its join attribute, any value from the domain underlying the join attribute. This assumption eliminates the need to have other parameters to characterize the data model, such as the number of distinguished values in both the source and target relations' join attribute. With this assumption the value of the latter parameters (as is shown later) can be obtained from the values of the parameters NTS, NTT and ND.

2. All the hashing functions, which are used in an equi-join algorithm, are statistically independent of each other.

3. All the hashing functions, which are used in an equi-join algorithm, are ideal. A hashing function is ideal if it is equally likely to map a value from its domain to any value of its range.

In Section 4.2.1 the equations which relate the performance parameter of every Basic algorithm to its execution model input parameters are developed. In Sections 4.2.2, 4.2.3, and 4.2.4 the latter process is repeated for the other

categories of the equi-join algorithms, namely, TPF, STPF and STCF.

4.2.1. Execution Models for the Basic Equi-Join Algorithms

The execution of a basic equi-join algorithm by RDBM can be decomposed into a number of similar phases. For the "global broadcast" basic algorithms the corresponding number of phases is equal to that of the MAUs which store the tuples of the source relation. For the "global hash" basic algorithms the corresponding number of phases is equal to that of the global buckets.

In order to calculate the TTIME the following notation is used:

K = the number of phases in the execution of a basic equi-join algorithm,

$x \in \{1,2,\ldots,K\}$,

TTI = total input time (time to move the tuples of the source and target relations from the mass storage to the PC's triplets),

TTI(x) = input time of phase x,

TTO = total output time (time to move the MAUs of the output relation from the PC's triplets to the moving-head disk),

TTO(x) = output time of phase x,

TTP = total triplet processor time (time for a triplet processor to execute the equi-join operation),

TTP(x) = time for a triplet processor to execute phase x,

TTPM(x) = time spent by a triplet processor during phase x in moving tuples within its LMU,

TTPH(x) = time taken by a triplet processor during phase x to hash tuples,

TTPC(x) = time taken by a triplet processor during phase x to compare the tuples' join attribute values,

TTPS(x) = time taken by a triplet processor to sort the source tuples of phase x,

TTB = total transmission time (time to move tuples across the cluster's TBUS during the execution of the equi-join operation),

TTB(x) = transmission time of phase x,

NTO = the expected number of tuples in the output relation,

NMS = the number of MAUs which store the source relation,

NMT = the number of MAUs which store the target relation,

NMO = The expected number of MAUs which store the output relation,

NGB = the number of global buckets.

The TTIME spent in executing one of the basic equi-join algorithms can be expressed as follows:

$$TTIME = TTO + \sum_{x=1}^{K} [TTI(x) + TTP(x) + TTB(x)]$$

Since TTI(x), TTP(x) and TTB(x) for all $x \varepsilon \{1,2, \ldots, k\}$ are on the average equal to TTI(1), TTP(1) and TTB(1), respectively, then

$$TTIME = TTO + K \cdot [\ TTI(1) + TTP(1) + TTB(1)\ ] \qquad (4.1)$$

In the following paragraphs, the formulas which compute the quantities TTO, TTI(1), TTP(1) and TTB(1) for the different basic equi-join algorithms are derived.

(a) TTO Derivation

The total output time TTO for each of the basic algorithms can be computed using the following formula:

$$TTO = NMO.(TBT + 2 \cdot TDAC + TDT) \qquad (4.2)$$

This formula states that for every MAU of the result relation it takes TBT to move the MAU to the parallel buffer block, TDAC to randomly locate an empty cylinder on the moving-head disk, TDT to move the MAU to that cylinder and TDAC to resume the transfer of the target MAUs.

In Section 1 of Appendix A the following expression for NMO is derived:

$$NMO = \left\lceil \frac{NTS.NTT.(LTS+LTT)}{ND.MAUC} \right\rceil$$

where " ⌈ ⌉ " denotes the ceiling function.

(b) TTI(1) Derivation

Two expressions exist for TTI(1), one for the "global broadcast" basic algorithms and the other for the " global hash " basic algorithms. In deriving these two expressions it is assumed that the MAUs which store the tuples of the source relation are stored in adjacent cylinders of the moving-head disk. The same assumption also applies to the target relation.

During one phase of a "global broadcast" basic algorithm one MAU of the source relation is joined with all the MAUs of the target relation. Therefore,

$$TTI(1) = (TDAC + TDT + TBT) + (TDAC + TDT + TBT) + (NMT - 1)(TSK + TDT + TBT) \qquad (4.3)$$

Moving a source MAU from the disk to the triplets' LMUs needs (on the average) TDAC to locate the cylinder of the moving-head disk which stores the source MAU, TDT to transfer the MAU to the parallel buffer block and TBT to transfer the MAU to the cluster's triplets. The movement of the first target MAU to the cluster's triplets takes the same time as that of the source MAU. Then for every remaining target MAU it takes TSK to move the disk heads to the next cylinder, TDT to transfer the cylinder content to the paral-

lel buffer block and TBT to transfer the content of the parallel buffer block to the cluster's triplets.

During one phase of a "global hash" basic algorithm the MAUs which store both the source and target relations are read off the moving-head-disk to the triplets' LMUs. This is needed in order to extract the set of source and target tuples which hash to the global bucket being processed by the PC . Therefore,

$$TTI(1) = (TDAC + TDT + TBT) + (NMS - 1)(TSK + TDT + TBT) + (TDAC + TDT + TBT) + (NMT - 1)(TSK + TDT + TBT) \quad (4.4)$$

(c) TTB(1) Derivation

Four expressions exist for TTB(1), one for the "global broadcast-local broadcast" basic algorithms, one for the "global broadcast-local hash" basic algorithms, one for the "global hash-local broadcast" basic algorithms and one for the "global hash-local hash" basic algorithms.

During one phase of a "global broadcast" basic algorithm the tuples of one MAU of the source relation are Joined with all the tuples of the target relation. To do this a "global broadcast-local broadcast" basic algorithm broadcasts every tuple in all the MAUs which store the target relation to all the cluster's triplets for joining it with the source tuples. Therefore,

$$TTB(1) = NTT.TT \quad (4.5)$$

To join one MAU of the source relation and all the MAUs of the target relation, a "global broadcast-local hash" algorithm will first redistribute the tuples of these MAUs, using a hashing function, among the cluster's triplets. Therefore,

$$TTB(1) = \left[\frac{NTS\ .TS}{NMS} + NTT.TT\right]\left[1-\frac{1}{NP}\right] \qquad (4.6)$$

where (NTS/NMS) is the average number of tuples in the source MAU and $(1-1/NP)$ is the average fraction of a triplet's tuples which need to be transferred to other triplets. In deriving the latter fraction we took advantage of both the first and the third basic assumptions stated earlier.

During one phase of a "global hash" basic algorithm the tuples of one global bucket are Joined. Using the first and the third basic assumptions one can easily find that the average number of the source and target relations which hash to a global bucket are (NTS/NGB) and (NTT/NGB) respectively.

To join the tuples of a global bucket, a "global hash-local broadcast" algorithm will broadcast the target relation tuples of the global bucket to the cluster triplets. Therefore,

$$TTB(1) = \frac{NTT}{NGB}.TT \qquad (4.7)$$

To join the tuples of a global bucket, a "global hash-local hash" algorithm will first redistribute the source and target tuples of the global bucket among the triplets. Therefore,

$$TTB(1) = \left[\frac{NTS}{NGB}\cdot TS + \frac{NTT}{NGB}\cdot TT\right]\left[1 - \frac{1}{NP}\right] \qquad (4.8)$$

(d) TTP(1) Derivation

In general, the $TTP(1)$ for a basic equi-join algorithm can be expressed as follows:

$$TTP(1) = TTPM(1) + TTPH(1) + TTPS(1) + TTPC(1) \qquad (4.9)$$

In the following sections, the formulas which compute the quantities $TTPM(1)$, $TTPH(1)$, $TTPS(1)$ and $TTPC(1)$ are derived for the different basic equi-join algorithms.

(1) TTPM(1) Derivation

In general, a triplet processor, executing a basic equi-join algorithm moves two types of data, namely, the output tuples and some source tuples. The output tuples, generated as a result of the joining process, are moved to BUFO. The average number of the output tuples is the same for all the Equi-Join Basic algorithms. In Section 1 of Appendix A, the following expression for the expected number of tuples in the output relation is derived:

$$\left\lceil \frac{NTS \cdot NTT}{ND} \right\rceil$$

Assuming that the generation of the output tuples is uniform over the PC's triplets and the execution phases of the equi-join algorithm, then the number of output tuples generated by one triplet during the execution of one phase of an equi-join algorithm is:

$$\frac{1}{K \cdot NP} \cdot \left\lceil \frac{NTS \cdot NTT}{ND} \right\rceil$$

Therefore, the time spent by one triplet to move the output tuples generated during one phase is:

$$\frac{1}{K \cdot NP} \left\lceil \frac{NTS \cdot NTT}{ND} \right\rceil \cdot TMO$$

In addition to the output tuples, a triplet processor executing a basic equi-join algorithm moves some of the source tuples within its local memory unit. The amount of tuples moved during the execution of one phase of an equi-join algorithm largely depends on its implementation details. The

tuples' movement is a time-consuming operation and a smart implementation of an equi-join algorithm is one which minimizes the amount of such movement. In the following calculation such implementation is assumed.

For a triplet processor executing one phase of a "global broadcast-local broadcast" basic algorithm, the tuples of the source MAU and the target MAUs can be read directly into BUFS and BUFI, respectively. Therefore, no tuples need to be moved by the triplet processor and the quantity TTPM(1) can be expressed as follows:

$$TTPM(1) = \frac{1}{K \cdot NP} \left\lceil \frac{NTS \cdot NTT}{ND} \right\rceil \cdot TMO \qquad (4.10)$$

For a triplet processor executing one phase of a "global broadcast-local hash" basic algorithm, a source MAU and the target MAUs are read, one at a time, into the PC's buffers BUFI. A triplet processor needs to move to BUFS only those tuples of the source MAU which hash to the corresponding triplet. The rest of the source MAU tuples are moved by the data mover. No target tuples need to be moved by the triplet processor. Therefore,

$$TTPM(1) = \frac{1}{K \cdot NP} \left\lceil \frac{NTS \cdot NTT}{ND} \right\rceil \cdot TMO + \frac{NTS}{NMS} \cdot \frac{1}{NP^2} TMS \qquad (4.11)$$

For a triplet processor executing one phase of a "global hash" basic algorithm, the MAUs which store the source and target relations are read, one at a time, into the BUFIs of the triplets. When executing one phase of a "global hash-local broadcast" basic algorithm a triplet processor needs only to move to BUFS those tuples of the source relation which hash to the current global bucket. No target tuples need to be moved by the triplet processor. Therefore,

$$TTPM(1) = \frac{1}{K \cdot NP} \left\lceil \frac{NTS \cdot NTT}{ND} \right\rceil \cdot TMO + CH \cdot \frac{NTS}{NGB} \cdot \frac{1}{NP} TMS \quad (4.12)$$

where CH has the value zero if the number of global buckets is one, otherwise, CH is one.

When executing one phase of a "global hash-local hash" basic algorithm a triplet processor needs only to move to BUFS those tuples of the source relation which hash to itself and to the current global bucket. On the other hand, no target tuples need to be moved by the triplet processor. Therefore,

$$TTPM(1) = \frac{1}{K.NP} \left\lceil \frac{NTS \cdot NTT}{ND} \right\rceil \cdot TMO + \frac{NTS}{NGB} \cdot \frac{1}{NP^2} \cdot TMS \quad (4.13)$$

(2) TTPH(1) Derivation

During the execution of one phase of a "global broadcast" basic algorithm, one MAU of the source relation and all the MAUs of the target relation are joined. A triplet processor executing one phase of the BBH-Basic algorithm must first hash and store its share of the source MAU tuples in the hash table. The triplet processor must also calculate the hashing function for its share of the target tuples. Therefore, for the *BBH algorithm*,

$$TTPH(1) = \left\lceil \frac{NTS}{NMS \cdot NP} + \frac{NTT}{NP} \right\rceil \cdot TH \quad (4.14)$$

Both the *BBC-Basic and the BBS-Basic algorithms* do not use any hashing technique. Therefore,

$$TTPH(1) = 0 \quad (4.15)$$

A triplet processor executing one phase of the BHC-Basic or the BHS-Basic algorithm must hash its share of tuples from one source MAU and all the target MAUs to the cluster's triplets. Therefore ,for the *BHC-Basic or the BHS-Basic algorithm*,

$$TTPH(1) = \left\lceil \frac{NTS}{NMS \cdot NP} + \frac{NTT}{NP} \right\rceil \cdot TH \qquad (4.16)$$

In addition to the above, a triplet processor executing one phase of the BHH-Basic algorithm must hash its share of tuples from one source MAU and all the target MAUs to the triplet's hash table. Therefore, for the *BHH-Basic algorithm*

$$TTPH(1) = 2 \cdot \left\lceil \frac{NTS}{NMS \cdot NP} + \frac{NTT}{NP} \right\rceil \cdot TH \qquad (4.17)$$

During the execution of one phase of a "global hash" basic algorithm, a hashing function must be computed for all the tuples of the source and target relations. This is needed in order to select the source and target tuples which hash to the current global bucket. This is the only hashing technique used by the *HBC-Basic and the HBS-Basic* algorithms. Therefore, for these two algorithms $TTPH(1)$ can be expressed as follows:

$$TTPH(1) = CH \cdot \left\lceil \frac{NTS}{NP} + \frac{NTT}{NP} \right\rceil \cdot TH \qquad (4.18)$$

where CH has the same definition as that of Equation (4.12).

In addition to the above, a triplet processor, executing one phase of the HBH-Basic algorithm, must calculate the hash-table hashing function for its share of the global bucket tuples. Therefore, for the *HBH-Basic algorithm*,

$$TTPH(1) = CH \cdot \left\lceil \frac{NTS}{NP} + \frac{NTT}{NP} \right\rceil \cdot TH + \left\lceil \frac{NTS}{NGB \cdot NP} \right\rceil \cdot TH \qquad (4.19)$$

During the execution of one phase of a "global hash-local hash" basic algorithm, two hashing functions are computed, one for all the tuples of the source and target relations and the other for those tuples of the source and target relations which hash to the global bucket. The latter hashing is used to distribute the tuples of the global bucket among the cluster's triplets for

processing. This is the only hashing technique used by the *HHC-Basic and HHS-Basic* algorithms. Therefore, for these two algorithms TTPH(1) can be expressed as follows:

$$TTPH(1) = CH \cdot \left[\frac{NTS}{NP} + \frac{NTT}{NP}\right] \cdot TH + \left[\frac{NTS}{NGB \cdot NP} + \frac{NTT}{NGB \cdot NP}\right] \cdot TH \quad (4.20)$$

In addition to the above, a triplet processor executing one phase of an HHH-Basic algorithm must calculate the "hash-table" hashing function for its share of tuples from the source and target relations. Therefore, for *HHH-Basic algorithm*,

$$TTPH(1) = CH \cdot \left[\frac{NTS}{NP} + \frac{NTT}{NP}\right] \cdot TH + 2 \cdot \left[\frac{NTS}{NGB \cdot NP} + \frac{NTT}{NGB \cdot NP}\right] \cdot TH \quad (4.21)$$

(3) TTPC(1) Derivation

In general the quantity TTPC(1) of a basic equi-join algorithm can be computed from the following equation:

$$TTPC(1) = ENC(1) \cdot T \quad (4.22)$$

where ENC(1) is the expected number of tuple comparisons performed by a triplet processor during the execution of one phase of the basic equi-join algorithm and T has the value of the parameter TCD for those basic algorithms which use the complete comparison technique, otherwise, T has the value of the parameter TCI.

Recall that during the execution of one phase of a "global broadcast" basic algorithm the tuples of a source MAU are joined with the tuples of all the target MAUs. To do so, the PC executing one phase of a "global broadcast-local broadcast" basic algorithm will broadcast all the target tuples, one tuple at a time, to the cluster triplets. For a triplet processor executing one phase of the *BBC-Basic algorithm*, ENC(1) can be expressed as

follows:

$$ENC(1) = NTT \cdot \frac{NTS}{NMS \cdot NP} \tag{4.23}$$

In Section 2a of Appendix A the following ENC(1) expression for a triplet processor executing one phase of the *BBH-Basic algorithm is derived:*

$$ENC(1) = \frac{NTT \cdot NTS}{NMS \cdot NP \cdot NBP} \left\{ \frac{1 - e^{-\frac{NTS_P}{ND}} e^{-\frac{NDS_P}{NBP}}}{1 - e^{-\frac{NDS_P}{NBP}}} \right\} \tag{4.24}$$

where

$$NTS_P = \frac{NTS}{NMS \cdot NP}$$

and

$$NDS_P = ND\left[1 - e^{-\frac{NTS_P}{ND}}\right]$$

In general, a triplet using the sorting method for joining a target tuple with a number of source tuples will first use the binary search method [KNUT73] to locate the address of the link list which stores those source tuples whose join attribute values match that of the target one. The number of comparisons, in the worst case, the triplet must perform is $\lceil \log_2 N \rceil$, where N is the number of the source tuples in that triplet and "$\lceil\ \rceil$" is the ceiling function.

Using the above argument, one can easily show that the $ENC(1)$ for the BBS-Basic algorithm can be expressed as follows:

$$ENC(1) = NTT \left\lceil \log_2 \frac{NTS}{NMS \cdot NP} \right\rceil \tag{4.25}$$

In general, during the execution of one phase of a "global broadcast-local hash" basic algorithm, the expected number of target tuples which hash to a

triplet processor for joining is $\left\lceil \frac{NTT}{NP} \right\rceil$. For a triplet processor executing one phase of the *BHC-Basic algorithm*, ENC(1) can be expressed as follows:

$$ENC(1) = \frac{NTT}{NP} \cdot \frac{NTS}{NMS \cdot NP} \tag{4.26}$$

In Section 2b of Appendix A the following ENC(1) expression for a triplet processor executing one phase of *the BHH-Basic algorithm is derived:*

$$ENC(1) = \frac{NTT \cdot NTS}{NMS \cdot NP^2 \cdot NBP} \left\{ \frac{1 - e^{-\frac{NTS_M}{ND}} e^{-\frac{NDS_P}{NBP}}}{1 - e^{-\frac{NDS_P}{NBP}}} \right\} \tag{4.27}$$

where

$$NTS_M = \frac{NTS}{NMS}$$

and

$$NDS_P = \frac{ND}{NP} \left\lceil 1 - e^{-\frac{NTS_M}{ND}} \right\rceil$$

For a triplet processor executing one phase of the *BHS-Basic algorithm*, ENC(1) can be expressed as follows:

$$ENC(1) = \frac{NTT}{NP} \left\lceil \log_2 \frac{NTS}{NMS \cdot NP} \right\rceil \tag{4.28}$$

Recall that, during the execution of one phase of a "global hash" Basic algorithm, the source and target tuples of a global Bucket are joined. The expected number of the source and target tuples which hash to a global bucket, during one the execution of phase, are $\left\lceil \frac{NTS}{NGB} \right\rceil$ and $\left\lceil \frac{NTT}{NGB} \right\rceil$ respectively. A triplet processor executing one phase of a "global hash-local broadcast" Basic algorithm will be, on the average, assigned $\left\lceil \frac{NTS}{NGB \cdot NP} \right\rceil$ tuples of

the source relation. The triplet processor will also process the $\left\lceil \frac{NTT}{NGB} \right\rceil$ target tuples against the assigned source tuples.

For a triplet processor executing one phase of the *HBC-Basic algorithm*, ENC(1) can be expressed as follows:

$$ENC(1) = \frac{NTT}{NGB} \cdot \frac{NTS}{NGB \cdot NP} \tag{4.29}$$

In Section 2c of Appendix A, the following ENC(1) expression for a triplet processor executing one phase of the *HBH-Basic algorithm is derived:*

$$ENC(1) = \frac{NTT \cdot NTS}{NP \cdot NBP \cdot NGB^2} \left\{ \frac{1 - e^{-\frac{NTS_S}{ND}} e^{-\frac{NDS_P}{NBP}}}{1 - e^{-\frac{NDS_P}{NBP}}} \right\} \tag{4.30}$$

where

$$NDS_P = \frac{ND}{NGB} \left[1 - e^{-\frac{NTS_S}{ND}} \right]$$

and

$$NTS_s = \frac{NTS}{NP}$$

For a triplet processor executing one phase of the *HBS-Basic algorithm*, $ENC(1)$ can be expressed as follows:

$$ENC(1) = \frac{NTT}{NGB} \left\lceil \log_2 \frac{NTS}{NGB \cdot NP} \right\rceil \tag{4.31}$$

A triplet processor executing one phase of a "global hash-local hash" basic algorithm will be, on the average, processing $\left\lceil \frac{NTS}{NGB \cdot NP} \right\rceil$ source tuples and $\left\lceil \frac{NTT}{NGB \cdot NP} \right\rceil$ target tuples. For a triplet processor executing one phase of the *HHC-Basic algorithm*, $ENC(1)$ can be expressed as follows:

$$ENC(1) = \frac{NTT}{NGB \cdot NP} \cdot \frac{NTS}{NGB \cdot NP} \tag{4.32}$$

In Section 2d of Appendix A, the following $ENC(1)$ expression for a triplet processor executing one phase of *the HHH-Basic algorithm is derived:*

$$ENC(1) = \frac{NTT \cdot NTS}{NGB^2 \cdot NP^2 \cdot NBP} \left\{ \frac{1 - e^{-\frac{NTS}{ND}} e^{-\frac{NDS_P}{NBP}}}{1 - e^{-\frac{NDS_P}{NBP}}} \right\} \tag{4.33}$$

where

$$NDS_P = \frac{ND}{NP \cdot NGB} \left\{ 1 - e^{\frac{NTS}{ND}} \right\}$$

For a triplet processor executing one phase of the HHS-Basic algorithm, $ENC(1)$ can be expressed as follows:

$$ENC(1) = \frac{NTT}{NGB \cdot NP} \left\lceil \log_2 \frac{NTS}{NGB \cdot NP} \right\rceil \tag{4.34}$$

(4) TTPS(1) Derivation

The quantity $TTPS(1)$ has a nonzero value for only those basic algorithms which use the sorting technique in performing the equi-join operation. In general, a triplet will internally sort M tuples of the source relation in, on the average, ($M \log_2 M$) comparisons and pointer swaps. Thus a general formula for $TTPS(1)$ would be:

$$TTPS(1) = (TCI + TEP) M \log_2 M \tag{4.35}$$

During the execution of one phase of the BBS-Basic or BHS-Basic algorithms, the expected number of source tuples assigned to a triplet is $\left\lceil \frac{NTS}{NMS \cdot NP} \right\rceil$. Substituting the latter quantity for M in Equation (4.35) yields the following equation for $TTPS(1)$ of both *the BBS-Basic and BHS-Basic algorithms:*

$$TTPS(1) = (TCI + TEP)\frac{NTS}{NMS \cdot NP}\left\lceil \log_2 \frac{NTS}{NMS \cdot NP} \right\rceil \qquad (4.36)$$

During the execution of one phase of the HBS-Basic or the HHS-Basic algorithm, the expected number of source tuples assigned to a triplet is $\left\lceil \frac{NTS}{NGB \cdot NP} \right\rceil$. Substituting the latter quantity for M in Equation (4.35) yields the following equation for $TTPS(1)$ of both the *HBS−Basic* and *HHS−Basic* algorithms:

$$TTPS(1) = (TCI + TEP)\frac{NTS}{NGB \cdot NP}\left\lceil \log_2 \frac{NTS}{NGB \cdot NP} \right\rceil \qquad (4.37)$$

4.2.2. Execution models for the TPF Equi-Join Algorithms.

Recall that every TPF equi-join algorithm is an extension of one of the basic equi-join algorithms. Similar to the basic equi-join algorithm, the execution of a TPF algorithm goes through a sequence of similar phases. The number of these phases is equal to that of a basic equi-join algorithm. During the execution of one phase of a basic equi-join algorithm, some tuples of the source and target relations are selected and joined. However, during the execution of one phase of a TPF algorithm and before joining the selected tuples of the source and target relations, a vector, initialized and maintained by the CMP, and a hashing function are used to encode the join attribute values of the selected source tuples. This vector is used to filter out many of the selected target tuples which do not match the source ones. The filtered out target tuples need not be processed any further.

From the preceding discussion it is easy to conclude that Equation (4.1) can be used to compute the *TTIME* spent in executing one algorithm of the TPF equi-join algorithms. Also, the formulas of Section 4.2.1 which compute the various quantities, other than $TTB(1)$, $TTPH(1)$ and $TTPC(1)$ for the basic

equi-join execution models can also compute the same quantities for each of the TPF equi-join execution models.

In the following sections, the formulas which compute the quantities $TTB(1)$, $TTPH(1)$ and $TTPC(1)$ for the various TPF equi-join execution models are derived.

(a) TTB(1) Derivation

Recall that during the execution of one phase of a "global broadcast" basic algorithm the tuples of one MAU of the source relation are joined with all the tuples of the target ones. In doing that, a "global broadcast-local broadcast" basic algorithm will broadcast every tuple of the target relation to the PC's triplets. On the other hand, a "global broadcast-local broadcast" TPF algorithm will broadcast only those tuples of the target relation which survive the vector checking.

Let

LTB be the fraction of the target relation tuples which survive the vector checking during the execution of one phase of a "global broadcast" TPF algorithm.

Then for a *"global broadcast-local broadcast" TPF equi-join algorithm*, $TTB(1)$ is:

$$TTB(1) = LTB \cdot NTT \cdot TT \tag{4.38}$$

In Section 3a of Appendix A, the following expression for LTB is derived:

$$LTB = 1 - e^{-\frac{NTS_M}{ND}} e^{-\frac{NDS_M}{NBIT}} \tag{4.39}$$

where

$$NTS_M = \frac{NTS}{NMS}$$

and

$$NDS_M = ND \cdot \left[1 - e^{-\frac{NTS_M}{ND}}\right]$$

To join one MAU of the source relation with all the MAUs of the target relation, a "global broadcast-local hash" TPF algorithm will first redistribute the tuples of the source MAU as well as those tuples of the target MAUs which survive the vector checking. Therefore,

$$TTB(1) = \left[\frac{NTS}{NMS} TS + LTB \cdot NTT \cdot TT\right]\left[1 - \frac{1}{NP}\right] \tag{4.40}$$

where LTB is computed from Equation (4.39)

Recall that during the execution of one phase of a "global hash" basic algorithm the tuples of one global bucket are joined. In doing that a "global hash-local broadcast" basic algorithm will broadcast the target tuples of the global bucket to the cluster triplets. However, a "global hash-local broadcast" TPF algorithm will broadcast only those target tuples of the global bucket which survive the vector checking. Therefore,

let

LTG be the fraction of the target relation tuples which hash to a global bucket and survive the vector checking during the execution of one phase of a "global hash" TPF algorithm.

Then for a "global hash-local broadcast" TPF equi-join algorithm, $TTB(1)$ is:

$$TTB(1) = LTG \cdot NTT \cdot TT \tag{4.41}$$

In Section 3b of Appendix A, the following expression for LTG is derived:

$$LTG = \frac{1}{NGB}\left[1 - e^{-\frac{NTS}{ND}} e^{-\frac{NDS_G}{NBIT}}\right] \tag{4.42}$$

where

$$NDS_G = \frac{ND}{NGB}\left[1-e^{-\frac{NTS}{ND}}\right]$$

To join the tuples of a global bucket a "global hash-local hash" TPF algorithm will redistribute the source tuples of the global bucket as well as the target tuples which hash to the global bucket and survive the vector checking. Therefore,

$$TTB(1) = \left[\frac{NTS}{NGB}\cdot TS + LTG\cdot NTT\cdot TT\right]\left[1-1/NP\right] \qquad (4.43)$$

where LTG is computed from Equation (4.42).

(b) TTPH(1) Derivation

During the execution of one phase of the *BBH−TPF* equi-join algorithm, a triplet processor will calculate two hashing functions. The first one, the hash-table function, is used to store the triplets' share of the source tuples into the triplets' hash tables. The second one, the hash-bit function, is used to encode the join attribute values of the latter tuples. In addition to that a triplet processor will calculate the two hashing functions for its share of the target tuples. Therefore, for the *BBH-TPF equi-join algorithm, TTPH(1) is:*

$$TTPH(1) = \left[\frac{NTS}{NMS\cdot NP} + \frac{NTT}{NP}\right]\cdot 2\cdot TH \qquad (4.44)$$

A triplet processor, executing one phase of the BBC-TPF or the BBS-TPF equi-join algorithm, will calculate only the hash-bit function for its share of both the source and the target tuples. Therefore, for a *BBC-TPF or BBS-TPF algorithm,*

$$TTPH(1) = \left[\frac{NTS}{NMS\cdot NP} + \frac{NTT}{NP}\right]\cdot TH \qquad (4.45)$$

A triplet processor executing one phase of the BHC-TPF or the BHS-TPF equi-join algorithm will compute two hashing functions for its share of tuples

from both the source MAU and all the target MAUs. The first one is the hash-bit function. The second one, the triplet-hash function, is used to hash the tuples to the PC's triplets. Therefore, for the *BHC-TPF or BHS-TPF algorithm*,

$$TTPH(1) = \left\lceil \frac{NTS}{NMS \cdot NP} + \frac{NTT}{NP} \right\rceil \cdot 2 \cdot TH \qquad (4.46)$$

In addition to calculating the above two hashing functions, a triplet processor executing one phase of BHH-TPF algorithm computes the hash-table function for its share of tuples from both the source MAU and from the surviving tuples of the target relation. Therefore, for the *BHH-TPF algorithm*,

$$TTPH(1) = \left\lceil \frac{NTS}{NMS \cdot NP} + \frac{NTT}{NP} \right\rceil \cdot 2 \cdot TH + \left\lceil \frac{NTS}{NMS \cdot NP} + \frac{LTB \cdot NTT}{NP} \right\rceil \cdot TH \qquad (4.47)$$

During the execution of one phase of a "global hash" TPF equi-join algorithm, a hashing function, the global-hash function, is computed for all the tuples of the source and target relations. This is needed in order to select the source and target tuples which belong to the current global bucket. The execution of one phase of the HBC-TPF or HBS-TPF algorithm will involve the additional computation of the hash-bit function for both the source and target tuples which hash to a global bucket. Therefore, for the *HBC-TPF or HBS-TPF algorithm*,

$$TTPH(1) = CH \cdot \left\lceil \frac{NTS}{NP} + \frac{NTT}{NP} \right\rceil \cdot TH + \left\lceil \frac{NTS}{NGB \cdot NP} + \frac{NTT}{NGB \cdot NP} \right\rceil \cdot TH \qquad (4.48)$$

where CH has the same definition as that of Equation (4.12).

In addition to calculating the above two hashing functions, a triplet processor executing one phase of the HBH-TPF equi-join algorithm calculates the hash-table function for its share of the source and target tuples which hash to a global bucket. Therefore, for the *HBH-TPF algorithm*,

$$TTPH(1) = CH \cdot \left[\frac{NTS}{NP} + \frac{NTT}{NP}\right] \cdot TH + \left[\frac{NTS}{NGB \cdot NP} + \frac{NTT}{NGB \cdot NP}\right] \cdot 2 \cdot TH \quad (4.49)$$

During the execution of one phase of the HHC-TPF or HHS-TPF algorithm, a triplet processor computes three hashing functions. The global-hash, the hash-bit and the triplet-hash functions. Therefore, for the *HHS-TPF or HHC-TPF algorithm*,

$$TTPH(1) = CH \cdot \left[\frac{NTS}{NP} + \frac{NTT}{NP}\right] \cdot TH + \left[\frac{NTS}{NGB \cdot NP} + \frac{NTT}{NGB \cdot NP}\right] \cdot 2 \cdot TH \quad (4.50)$$

In addition to calculating the above three hashing functions, a triplet processor executing one phase of the HHH-TPF algorithm, computes another hashing function, the hash-table function. A triplet processor computes the latter function for its share of the global bucket source tuples and its share of the target relation tuples which belong to the global bucket and survive the vector checking. Therefore, for the *HHH-TPF algorithm*,

$$TTPH(1) = CH \cdot \left[\frac{NTS}{NP} + \frac{NTT}{NP}\right] \cdot TH + \left[\frac{NTS}{NGB \cdot NP} + \frac{NTT}{NGB \cdot NP}\right] \cdot 2 \cdot TH + \left[\frac{NTS}{NGB \cdot NP} + \frac{LTG \cdot NTT}{NP}\right] \cdot TH \quad (4.51)$$

(c) TTPC(1) Derivation

In general, the quantity TTPC(1) of a TPF equi-join algorithm can be computed from the following equation:

$$TTPC(1) = ENC(1) \cdot T \quad (4.52)$$

where ENC(1) is the expected number of tuple comparisons performed by a triplet processor during the execution of one phase of a TPF equi-join algorithm and T has the value of the parameter TCD for those TPF algorithms which use the complete comparison technique, otherwise, T has the value of

the parameter TCI.

Recall that during the execution of one phase of a "global broadcast" TPF algorithm the tuples of a source MAU are joined with the tuples of all the target MAUs. To do so the PC, executing one phase of a "global broadcast-local broadcast" TPF algorithm, will broadcast all the target tuples which survive the vector checking to the cluster triplets. Thus for a triplet processor executing one phase of the *BBC-TPF algorithm*, ENC(1) can be expressed as follows:

$$ENC(1) = LTB \cdot NTT \cdot \frac{NTS}{NMS \cdot NP} \tag{4.53}$$

where LTB is computed from Equation (4.39).

For a triplet processor executing one phase of the BBS-TPF algorithm, ENC(1) can be expressed as follows:

$$ENC(1) = LTB \cdot NTT \left\lceil \log_2 \frac{NTS}{NMS \cdot NP} \right\rceil$$

In Section 3c of Appendix A, the following ENC(1) expression for a triplet processor executing one phase of the *BBH-TPF algorithm is derived:*

$$ENC(1) = \frac{NTT \cdot NTS}{NMS \cdot Np \cdot NBP} \left\{ \frac{1-e^{-\frac{NTS_P}{ND}} + (1-e^{-\frac{NDS_M}{NBIT}})(1-e^{-\frac{NDS_P}{NBP}})e^{-\frac{NTS_P}{ND}}}{1-e^{-\frac{NDS_P}{NBP}}} \right\} \tag{4.54}$$

where NTS_p, NDS_p and NDS_M are the same as those of Equations (4.25), (4.27) and (4.39), respectively.

In general, during the execution of a phase of a "global broadcast-local hash" TPF algorithm, the expected number of target tuples which survive the vector checking and hash to a triplet processor, for joining, is $\left\lceil \frac{LTB \cdot NTT}{NP} \right\rceil$.

Thus for a triplet processor executing one phase of the *BHC-TPF algorithm*, ENC(1) can be expressed as follows:

$$ENC(1) = \frac{LTB \cdot NTT}{NP} \frac{NTS}{NMS \cdot NP} \tag{4.55}$$

For a triplet processor executing one phase of the *BHS-TPF algorithm*, ENC(1) can be expressed as follows:

$$ENC(1) = \frac{LTB \cdot NTT}{NP} \left\lceil \log_2 \frac{NTS}{NMS \cdot NP} \right\rceil \tag{4.56}$$

In Section 3d of Appendix A the following ENC(1) expression for a triplet processor executing one phase of the *BHH-TPF algorithm is derived:*

$$ENC(1) = \frac{NTT \cdot NTS}{NMS \cdot NP^2 \cdot NBP} \left\{ \frac{1-e^{-\frac{NTS_M}{ND}} + (1-e^{-\frac{NDS_P}{NBP}})(1-e^{-\frac{NDS_M}{NBIT}})e^{-\frac{NTS_M}{ND}}}{1-e^{-\frac{NDS_P}{NBP}}} \right\} \tag{4.57}$$

where NTS_M and NDS_P are the same as those of Equation (4.27) NDS_M is the same as that of Equation (A.29).

Recall that during the execution of one phase of a "global hash" TPF algorithm, the source and target tuples of a global bucket are joined. The average number of the source and target tuples which hash to a global bucket are $\left\lceil \frac{NTS}{NGB} \right\rceil$ and $\left\lceil \frac{NTT}{NGB} \right\rceil$, respectively. The vector checking will reduce the target tuples of the global bucket to $(LTG \cdot NTT)$ where LTG can be computed using Equation (4.42). During the execution of one phase of a "global hash-local broadcast" TPF algorithm, a triplet processor is assigned, on the average, $\left\lceil \frac{NTS}{NGB \cdot NP} \right\rceil$ tuples of the source relation to process. Thus for a triplet processor executing one phase of the *HBC-TPF algorithm*, ENC(1) can be expressed

as follows:

$$ENC(1) = LTG \cdot NTT \cdot \frac{NTS}{NGB \cdot NP} \tag{4.58}$$

For a triplet processor executing one phase of the *HBS-TPF algorithm*, ENC(1) can be expressed as follows:

$$ENC(1) = LTG \cdot NTT \cdot \left\lceil \log_2 \frac{NTS}{NGB \cdot NP} \right\rceil \tag{4.59}$$

In Section 2e of Appendix A, the following ENC(1) expression for a triplet processor executing one phase of the *HBH-TPF algorithm is derived:*

$$ENC(1) = \frac{NTT \cdot NTS}{NGB^2 \cdot NP \cdot NBP} \left\{ \frac{1-e^{-\frac{NTS_S}{ND}} + (1-e^{-\frac{NDS_P}{NBP}})(1-e^{-\frac{NDS_G}{NBIT}})e^{-\frac{NTS_S}{ND}}}{1-e^{-\frac{NDS_P}{NBP}}} \right\} \tag{4.60}$$

where NTS_s, NDS_p and NDS_G are defined in Equations (A.30), (A.26) and (A.44), respectively.

During the execution of one phase of a "global hash-local hash" TPF algorithm, a triplet processor is assigned, on the average, $\left\lceil \frac{NTS}{NGB \cdot NP} \right\rceil$ tuples of the source relation. These tuples are compared with the target tuples which hash to the global bucket, survive the vector checking and hash to the triplet processor. The average number of the latter tuples is $\left\lceil \frac{LTG \cdot NTT}{NP} \right\rceil$. Thus for a triplet processor executing one phase of the *HHC-TPF algorithm*, ENC(1) can be expressed as follows:

$$ENC(1) = \frac{LTG \cdot NTT}{NP} \cdot \frac{NTS}{NGB \cdot NP} \tag{4.61}$$

For a triplet processor executing one phase of the *HHS-TPF algorithm*, ENC(1) can be expressed as follows:

$$ENC(1) = \frac{LTG \cdot NTT}{NP} \left\lceil \log_2 \frac{NTS}{NGB \cdot NP} \right\rceil \tag{4.62}$$

In Section 3f of Appendix A, the following ENC(1) expression for a triplet processor executing one phase of the *HHH-TPF algorithm is derived:*

$$ENC(1) = \frac{NTT \cdot NTS}{NGB^2 \cdot NP^2 \cdot NBP} \left\{ \frac{1-e^{-\frac{NTS}{ND}} + (1-e^{-\frac{NDS_P}{NBP}})(1-e^{-\frac{NDS_G}{NBIT}})e^{\frac{-NTS}{ND}}}{1-e^{-\frac{NDS_P}{NBP}}} \right\} \tag{4.63}$$

where NDS_P and NDS_G are defined as those of Equations (4.33) and (A.44), respectively.

4.2.3. Execution Models for the STPF Equi-Join Algorithms

Recall that every STPF equi-join algorithm is an extension of one of the "global hash" TPF equi-join algorithms. As for a TPF algorithm, the execution of an STPF goes through a sequence of similar phases. The number of these phases is equal to NGB, the number of global buckets. During the execution of one phase of a "global hash" TPF algorithm, the source and target tuples of one global bucket are joined. In the latter algorithms a vector (BIT-S), initialized and maintained by the cluster master processor, is used to encode the join attribute values of the source tuples which belong to the current global bucket. This memory will then be used to filter out many of the global bucket target tuples whose join attribute values do not match those of the encoded source ones. In addition to BIT-S, an STPF algorithm uses another vector (BIT-T), initialized and maintained by the cluster master processor, to encode the join attribute values of the target tuples which belong to the current glo-

bal bucket. BIT-T will then be used to filter out some of the source tuples which do not have matches among the encoded target ones.

From the above discussion, it is easy to conclude that Equation (4.1) can be used to compute the TTIME spent in executing one algorithm of the STPF equi-join algorithms. Also the formulas of Section 4.2.1 which compute the quantities TTO and TTI(1) for the "global Hash" TPF equi-join execution models can also be used to compute the same quantities for each of the STPF equi-join execution models.

In the following paragraph, the formulas which compute the quantities TTB(1), TTPM(1), TTPH(1), TTPC(1) and TTPS(1) for the various STPF equi-join execution models are derived.

(a) TTB(1) Derivation

During the execution of one phase of a "local broadcast" STPF algorithm, the cluster master processor broadcasts, to all the cluster's triplets, every target tuple which belongs to the global bucket and survives the BIT-S vector checking.

Let

$x \in \{S,T\}$

Lx be the fraction of relation x tuples which hash to a global bucket and survive the corresponding vector (BIT-S for target tuples, BIT-T for source tuples) checking.

Then for a *"local broadcast" STPF algorithm*, TTB(1) is:

$$TTB(1) = LT \cdot NTT \cdot TT \tag{4.64}$$

where the parameter LT is computed using the equation developed for the parameter LTG of the previous section. That is, LT can be expressed as follows:

$$LT = \frac{1}{NGB}\left\{1 - e^{-\frac{NTS}{ND}} e^{-\frac{NDS_G}{NBIT}}\right\} \tag{4.65}$$

where

$$NDS_G = \frac{ND}{NGB}\left[1 - e^{\frac{-NTS}{ND}}\right]$$

During the execution of one phase of a "local hash" STPF algorithm, the source and target tuples which hash to the global bucket and survive the corresponding vector checking must be redistributed. Therefore, for a *"local hash" STPF algorithm*, TTB(1) is:

$$TTB(1) = (\ LS \cdot NTS \cdot NT + LT \cdot NTT \cdot TT\ )(1 - 1/NP) \tag{4.66}$$

where LT is computed using Equation (4.65). A formula for computing LS can be derived by following the same steps used to derive Equation (4.65). However, the quantities NTS and NDS_G must be replaced by the quantities NTT and NDT_G, respectively. Thus LS has the following equation:

$$LS = \frac{1}{NGB}\left[1 - e^{-\frac{NTT}{ND}} e^{-\frac{NDT_G}{NBIT}}\right] \tag{4.67}$$

where

$$NDT_G = \frac{ND}{NGB}\left[1 - e^{-\frac{NTT}{ND}}\right]$$

(b) TTPM(1) Derivation

Similar to a TPF (or basic) equi-join algorithm, a triplet processor executing an STPF algorithm moves two types of data, namely, the output tuples and some source and target tuples. The output tuples, generated as a result of the joining process, are moved to BUFO. The time spent by one triplet processor to move its share of the output tuples generated during the execution of one phase of an STPF algorithm is computed using Equation (4.10).

During the execution of one phase of a "local broadcast" STPF algorithm, a triplet processor moves, to a temporary storage, its share of those target tuples which belong to the global bucket. The triplet processor also moves, to BUFS , its share of the source tuples which hash to the global bucket and survive the BIT-T checking. Therefore, for a triplet processor executing one phase of a *"local broadcast" STPF algorithm*, TTPM(1) is:

$$TTPM(1) = \frac{1}{K \cdot NP}\left\lceil \frac{NTS \cdot NTT}{ND} \right\rceil \cdot TMO + CH \cdot \frac{NTT}{NGB \cdot NP} \cdot TMT + \frac{LS \cdot NTS}{NP} \cdot TMS \qquad (4.68)$$

where CH has zero value if $NGB = 1$, otherwise, one.

During the execution of one phase of a "local hash" STPF algorithm, a triplet processor moves, to a temporary storage its share of those source tuples which hash to the global bucket, survive the BIT-T vector checking and do not need to be communicated to a different triplet. Therefore, for a triplet processor executing one phase of a *"local hash" STPF algorithm*, TTPM(1) is:

$$TTPM(1) = \frac{1}{K \cdot NP}\left\lceil \frac{NTS \cdot NTT}{ND} \right\rceil \cdot TMO + CH \cdot \frac{NTT}{NGB \cdot NP} \cdot TMT + \frac{LS \cdot NTS}{NP^2} \cdot TMS \qquad (4.69)$$

(c) TTPH(1) Derivation

During the execution of one phase of an STPF algorithm, the PC computes the global-hash function for every tuple of the source and target relations. This is needed in order to select the source and target tuples which hash to the current global bucket. The execution of one phase of the HBC-STPF or HBS-STPF algorithm will involve the additional computation of the hash-bit function for both the source and target tuples which belong to a

global bucket.* Therefore, for the *HBC-STPF or the HBS-STPF algorithm*,

$$TTPH(1) = CH \cdot \left[\frac{NTS}{NP} + \frac{NTT}{NP}\right] \cdot TH + \left[\frac{NTS}{NGB \cdot NP} + \frac{NTT}{NGB \cdot NP}\right] \cdot TH \quad (4.70)$$

In addition to computing the global-hash and the hash-bit functions, a triplet processor executing one phase of the HBH-STPF algorithm computes the hash-table function for its share of those source and target tuples of the global bucket which survive the corresponding vector checking (checking BIT-T for the source tuples, checking BIT-S for the target tuples). Therefore, for the *HBH-STPF algorithm*,

$$TTPH(1) = CH \cdot \left[\frac{NTS}{NP} + \frac{NTT}{NP}\right] \cdot TH + \left[\frac{NTS}{NGB \cdot NP} + \frac{NTT}{NGB \cdot NP}\right] \cdot TH + \left[\frac{LS \cdot NTS}{NP} + \frac{LT \cdot NTT}{NP}\right] \cdot TH \quad (4.71)$$

During the execution of one phase of the HHC-STPF or the HHS-STPF algorithm, a triplet processor computes three hashing functions, namely, the global-hash, the hash-bit and the triplet-hash. Therefore, for the *HHS-STPF or HHC-STPF algorithm*,

$$TTPH(1) = CH \cdot \left[\frac{NTS}{NP} + \frac{NTT}{NP}\right] \cdot TH + \left[\frac{NTS}{NGB \cdot NP} + \frac{NTT}{NGB \cdot NP}\right] \cdot 2 \cdot TH \quad (4.72)$$

In addition to computing the above three hashing functions, a triplet processor executing one phase of the HHH-STPF algorithm computes another hashing function, the hash-table function. A triplet processor computes the latter function for its share of the global bucket source and target tuples which survive the vector's checking. Therefore, for the *HHH-STPF algorithm*,

*The address of the bit, in BIT-T which encode the join attribute value of a given global bucket target tuple, is stored together with the corresponding tuple. This eliminates the needs to reevaluate the hash-bit hashing function for the target tuple of the current global bucket.

$$TTPH(1) = CH \cdot \left\lceil \frac{NTS}{NP} + \frac{NTT}{NP} \right\rceil \cdot TH + \left\lceil \frac{NTS}{NGB \cdot NP} + \frac{NTT}{NGB \cdot NP} \right\rceil \cdot 2 \cdot TH + \left\lceil \frac{LS \cdot NTS}{NP} + \frac{LT \cdot NTT}{NP} \right\rceil \cdot TH \tag{4.73}$$

(d) TTPC(1) Derivation

Recall that, during the execution of one phase of an STPF algorithm, the source and target tuples which hash to a global bucket and survive the vector's checking are $(LS \cdot NTS)$ and $(LT \cdot NTT)$, respectively. During the execution of one phase of a "local broadcast" STPF algorithm a triplet processor stores in BUFS, on the average, $\left\lceil \frac{LS \cdot NTS}{NP} \right\rceil$ tuples of the source relation. These tuples are compared with every target tuple of the global bucket which survives BIT-S checking. Thus for a triplet processor executing one phase of the *HBC-STPF algorithm*, ENC(1) can be expressed as follows:

$$ENC(1) = LT \cdot NTT \cdot \frac{LS \cdot NTS}{NP} \tag{4.74}$$

With Equation (4.35) in mind, it is easy to see that for the *HBS-STLF algorithm*, ENC(1) can be expressed as follows:

$$ENC(1) = LT \cdot NTT \cdot \left\lceil \log_2 \frac{LS \cdot NTS}{NP} \right\rceil \tag{4.75}$$

In Section 4a of Appendix A, the following ENC(1) expression for a triplet processor executing one phase of the *HBH-STPF algorithm is derived:*

$$ENC(1) = \frac{NTT \cdot NTS \cdot LS}{NP \cdot NGB \cdot NBP} \left\{ \frac{1 - e^{-\frac{NTS_S}{ND}} + e^{-\frac{NTS_S}{ND}} \left[1 - e^{-\frac{NDSF_P}{NBP}}\right] \left[1 - e^{-\frac{NDS_G}{NBIT}}\right]}{1 - e^{-\frac{NDSF}{NBP}P}} \right\} \tag{4.76}$$

During the execution of one phase of a "local hash" STPF algorithm, a triplet processor stores in BUFS, on the average, $\left\lceil \frac{LS \cdot NTS}{NP} \right\rceil$ tuples of the source relation to process. These tuples are compared with those target tuples of the global bucket which survive BIT-S checking and hash to a triplet processor. On the average, the number of these tuples is $\left\lceil LT \cdot \frac{NTT}{NP} \right\rceil$. Thus for a triplet processor executing one phase of the *HHC-STPF algorithm*, ENC(1) can be expressed as follows:

$$ENC(1) = \frac{LT \cdot NTT}{NP} \cdot \frac{LS \cdot NTS}{NP} \tag{4.77}$$

For the *HHS-STPF algorithm*, ENC(1) can be expressed as follows:

$$ENC(1) = \frac{LT \cdot NTT}{NP} \left\lceil \log_2 \frac{LS \cdot NTS}{NP} \right\rceil \tag{4.78}$$

In Section 4b of Appendix A, the following ENC(1) expression for a triplet processor executing one phase of the *HHH-STPF algorithm is derived:*

$$ENC(1) = \frac{NTT \cdot NTS \cdot LS}{NP^2 \cdot NGB \cdot NBP} \left\{ \frac{1 - e^{-\frac{NTS}{ND}} + e^{-\frac{NTS}{ND}} \left[1 - e^{-\frac{NDSF_P}{NBP}}\right] \left[1 - e^{-\frac{NDS_G}{NBIT}}\right]}{1 - e^{-\frac{NDSF_P}{NBP}}} \right\} \tag{4.79}$$

(e) TTPS(1) Derivation

The quantity TTPS(1) has a nonzero value for only two STPF algorithms, namely, the HBS-STPF and the HHS-STPF algorithms. Recall that Equation (4.35) computes the time a triplet processor spends in sorting M tuples. During the execution of one phase of the HBS-STPF or HHS-STPF algorithms, the expected number of source tuples assigned to a triplet (M) is $\left\lceil \frac{LS \cdot NTS}{NP} \right\rceil$. Sub-

stituting the latter quantity for M in Equation (4.35) yields the following equation for TTPS(1) for both the HBS-STPF and HHS-STPF algorithms:

$$TTPS(1) = (TCI + TMP)\frac{LS \cdot NTS}{NP}\left\lceil \log_2 \frac{LS \cdot NTS}{NP} \right\rceil \qquad (4.80)$$

4.2.4. Execution Models for the STCF Equi-Join Algorithms

In this section, the execution models are presented for every STCF algorithm except those algorithms which use the hash-table method in joining the tuples within a PC's triplet.

Recall that the execution of an STCF algorithm is comprised of two phases, namely, the global filtering phase and the join phase. The computation of the TTIME spent in executing an STCF algorithm involves the computation of the TTIME spent in executing both of the latter phases.

Let

TTIMEG be the TTIME spent in executing the global filtering phase of an STCF algorithm, and

TTIMEJ be the TTIME spent in executing the join phase of an STCF algorithm.

The TTIME spent in executing an STCF algorithm can then be expressed as follows:

$$TTIME = TTIMEG + TTIMEJ \qquad (4.81)$$

In the following paragraphs, the formulas which compute the quantities TTIMEG and TTIMEJ are derived.

(a) Derivations for TTIMEG

The global filtering phase is common to all the STCF algorithms. Therefore, the TTIMEG spent in executing this phase is the same for all the STCF

algorithms. Recall that the execution of a global filtering phase is comprised of three subphases. Therefore, the quantity TTIMEG is the sum of the TTIME spent in executing each of these phases.

Let

$$x \in \{1, 2, 3\}$$

and $TTIMEG(x)$ be the TTIME spent in executing subphase x of the global filtering phase.

Then, the TTIMEG can then be expressed as follows:

$$\begin{aligned} TTIMEG &= \sum_{x=1}^{3} TTIMEG(x) \\ &= \sum_{x=1}^{3} [TTI(x) + TTO(x) + TTP(x)] \end{aligned} \tag{4.82}$$

In the following sections, the equations which compute the quantities $TTI(x)$, $TTO(x)$, and $TTP(x)$ for all x are developed.

(1) TTI(x) Computation

During the execution of the first subphase of the global filtering phase, all the MAUs which store the target tuples are read, one MAU at a time, from the moving-head disk into the PC's LMUs. Therefore,

$$TTI(1) = (TDAC + TDT + TBT) + (NMT-1)(TSK + TDT + TBT) \tag{4.83}$$

During the execution of the second subphase, all the MAUs which store the source tuples are read, one MAU at a time, from the moving-head disk into the PC's LMUs.

Therefore,

$$TTI(2) = (TDAC + TDT + TBT) + (NMS-1)(TSK + TDT + TBT) \tag{4.84}$$

Finally, during the execution of the third subphase, all the MAUs which store the target tuples are again read, one MAU at a time, from the moving-head disk into the cluster's LMUs. Therefore,

$$TTI(3) = TTI(1) \tag{4.85}$$

(2) TTO(x) Computation

During the execution of the first subphase, no tuples need to be moved from the cluster's LMUs to the moving-head disk. Therefore,

$$TTO(1) = 0 \tag{4.86}$$

During the execution of the second subphase, the tuples of the target relation which survive BIT-S checking need to be moved, in MAU units, to the moving-head disk.

Let $y \in \{S,T\}$

NMyF be the number of MAUs which store those tuples of relation y which survive the corresponding vector checking.

Then, the TTO(2) can be expressed as follows:

$$TTO(2) = NMTF \cdot (2 \cdot TDAC + TDT + TBT) \tag{4.87}$$

It is easy to see that the quantity NMTF can be expressed as follows:

$$NMTF = \left\lceil \frac{LT \cdot NTT \cdot LTT}{MAUC} \right\rceil$$

In Section 5a of Appendix A, the following formula for LT is derived:

$$LT = 1 - e^{-\frac{NTS}{ND}} e^{-\frac{NDS}{NBIT}} \tag{4.86}$$

During the execution of the third subphase, the tuples of the source relation which survive BIT-T checking need to be moved, in MAU units, to the the moving-head disk. Therefore,

$$TTO(3) = NMSF \cdot (2 \cdot TDAC + TDT + TBT) \tag{4.89}$$

where the quantity NMSF can be computed from the following equation:

$$NMSF = \left\lceil \frac{LS \cdot NTS \cdot LTS}{MAUC} \right\rceil$$

In Section 5b of Appendix A, the following formula for LS is derived:

$$LS = 1 - e^{-\frac{NTT}{ND}} e^{-\frac{NDT}{NBIT}} \tag{4.90}$$

(3) TTP(x) Computation

During the execution of the first subphase, a triplet processor computes the hash-bit function for its share of the source tuples. Therefore,

$$TTP(1) = \frac{NTS}{NP} \cdot TH \tag{4.91}$$

During the execution of the second subphase, a triplet processor computes the hash-bit function for its share of the target tuples. Also the triplet processor moves, to the output buffer, those tuples of the target relation which survive BIT-S checking. Therefore, the quantity TTP(2) can be expressed as follows:

$$TTP(2) = \frac{NTT}{NP} \cdot TH + \frac{LT \cdot NTT}{NP} \cdot TMT \tag{4.92}$$

During the execution of the third subphase, a triplet processor computes the hash-bit function for its share of the source tuples. Also the triplet processor moves, to the output buffer, those tuples of the source relation which survive BIT-T checking. Therefore, the quantity TTP(3) can be expressed as follows:

$$TTP(3) = \frac{NTS}{NP} \cdot TH + \frac{LS \cdot NTS}{NP} \cdot TMS \tag{4.93}$$

(b) Derivation for TTIMEJ

During the join phase of the STCF algorithms, the source and target tuples are joined using one of the basic equi-join algorithms. .Therefore, the computation of TTIMEJ for executing the join phase of an STCF algorithm is similar to that of computing TTIME for the corresponding basic equi-join algorithm.* The quantity TTIMEJ can be computed using Equation (4.4). That is, TTIMEJ can be expressed as follows:

$$TTIMEJ = TTO + K \cdot [TTI(1) + TTO(1) + TTP(1) + TTB(1)] \qquad (4.94)$$

Also the quantity TTP(1) can be expressed as follows:

$$TTP(1) = TTPM(1) + TTPH(1) + TTPC(1) + TTPS(1) \qquad (4.95)$$

The equations which were developed for computing the quantities K, TTO, $TTI(1)$, $TTB(1)$, $TTPM(1)$, $TTPH(1)$, $TTPC(1)$ and $TTPS(1)$ of the basic execution models can also be used, with slight modification, to compute the same quantities of Equations (4.94) and (4.95). The equations developed for the basic execution models must be modified to include the effect of the global filtering phase. The parameters NTS and NTT of the equations generated for the basic execution models must be replaced by the parameters NTSF and NTTF, respectively, where

$$NTSF = LS \cdot NTS \qquad (4.96)$$

and

$$NTTF = LT \cdot NTT \qquad (4.97)$$

*This statement is true only for those STCF algorithms which do not use the hash-table method. This section, therefore, only models the execution of the STCF algorithms and corresponds to the basic ones which satisfy the latter statement.

4.3. The Evaluation of the Proposed Equi-Join Algorithms

In this section, the performance of the different equi-join algorithms, proposed for the new RDBM, are studied and compared. This is carried out in two steps. In the first step, each of the equi-join algorithmic categories (basic, TPF, STPF and STCF) are separately studied using the corresponding execution models already developed in the previous section. In this step, the best performing algorithm(s) within each of these categories is also determined.

In the second step, the best performing algorithms from each of the equi-join algorithmic categories are compared. The overall best performing algorithm(s) under the different data environment is then determined.

Two basic factors are considered in selecting the best performing algorithm, namely, the performance measure TTIME and the number of hashing functions. In general, the best performing algorithm, in a set containing others, over a range of a parameter' values is the one with the smallest values of TTIME over that range. However, if more than one algorithm has close TTIME values over that range, then the best performing algorithm is the one with a minimum number of hashing functions. The last criterion is adopted to compensate for the fact that in our performance evaluation we considered only the average behavior of the hashing functions, not the worst case one; moreover we did not consider the overhead involved in handling the overflow phenomena associated with their use [KNUT73]. Therefore, it is argued that in order for an algorithm with a hashing function to be adopted for executing the equi-join operation, it must show a significant performance improvement over the one with no hashing function.

In Appendix B, the input parameters of the equi-join models are classified into two groups, namely, the static parameters and the dynamic parameters. The static parameters are those whose values are kept constant throughout

the evaluation of the equi-join execution models. The values selected for the static parameters together with the reasons behind such selection are presented in Appendix B.

The dynamic parameters are NTS, NTT, ND, NBP and NBIT. To reduce the number of dynamic parameters it is assumed that both NTS and NTT have the same value. ND will be changed indirectly through changing the parameter (NTS/ND). In Appendix B it is shown that, throughout the performance evaluation, one can assign values to the parameters NBP and NBIT using the following formulas:

$$NBP = \begin{cases} Min\left\{\frac{NDD_{BP}}{4}, \left\lfloor FNBP \cdot \frac{NTS}{NMS \cdot NP}\right\rfloor\right\} & \text{for } \textit{the global broadcast algorithms} \\ Min\left\{\frac{NDD_{BP}}{4}, \left\lfloor FNBP \cdot \frac{NTS}{NGB \cdot NP}\right\rfloor\right\} & \text{for } \textit{the global hash algorithms} \end{cases}$$

$$NBIT = \begin{cases} Min\left\{\frac{NDD_{BI}}{4}, \lfloor FNBIT \cdot [\, NTS + NTT \,] \rfloor\right\} & \text{for } \textit{all algorithms with one CMP vector} \\ Min\left\{\frac{NDD_{BI}}{4}, \lfloor FNBIT \cdot [\, NTS + NTT \,]/2 \rfloor\right\} & \text{for } \textit{all algorithms with two CMP vectors} \end{cases}$$

where NDD_{BP} and NDD_{BI} are functions of ND (together with some other static parameters). FNBP and FNBIT are functions of the capacity of the storage allocated for the hash table within a triplet, and the vector, respectively.

In Appendix B, the range of values which the parameters NTS (NTT) and [NTS/ND] will assume throughout the performance evaluation is presented. The results of this evaluation are presented in the following sections.

4.3.1. The Evaluation of the Basic Equi-Join Algorithms

A computer program was written to compute the performance measure TTIME for every basic equi-join algorithm using the Equations already

developed in Section 4.2.1. The set of programs were run for NTS(NTT) ε $\{10^3 - 10^5\}$, NTS/ND ε {1, .1, .01} and FNBP ε {1, 5, 10}. The corresponding behavior of TTIME is shown in Figure (4.5).

From Figure (4.5), the following important conclusions can be drawn:

1. Within the range of values assumed for [NTS/ND], namely (1, .1, .01), the performance measure TTIME for all the basic equi-join algorithms is independent of the joining probability* values.

2. Within the range of values assumed for FNBP, namely (1, 5, 10), the performance measure TTIME, for all the basic algorithms which use the hash table method (BBH-Basic, BHH-Basic, HBH-Basic, HHH-Basic), is independent of the number of entries in the corresponding table.

3. The hashing functions improve the execution speed of the equi-join operation considerably. For example, joining two relations, each with 10^4 tuple, using the HBH-Basic algorithm is approximately 650 times faster than that using the BBC-Basic algorithm.

4. Executing the equi-join operation using an algorithm other than HBS-Basic either degrades the operation performance (as in the case of using the basic algorithms BBC, BBS, BBH, BHC, BHS, BHH, HBC and HHC) or does not improve the performance enough to justify its usage (as in the case of using the basic algorithms HBH, HHS and HHH). Thus the algorithm HBS-Basic is considered as the best performing one within the basic algorithmic category.

One final note regarding the basic equi-join algorithms. Recall that only the BBC-Basic and BBS-Basic algorithms can be used for executing the nonequi-Join operations. Figure (4.5) shows that the performance of the BBS-

*This is defined, in Appendix B, as the probability that a target tuple finds at least one source tuple with matching join attribute value. It is also shown that this probability is related to [NTS/NTT].

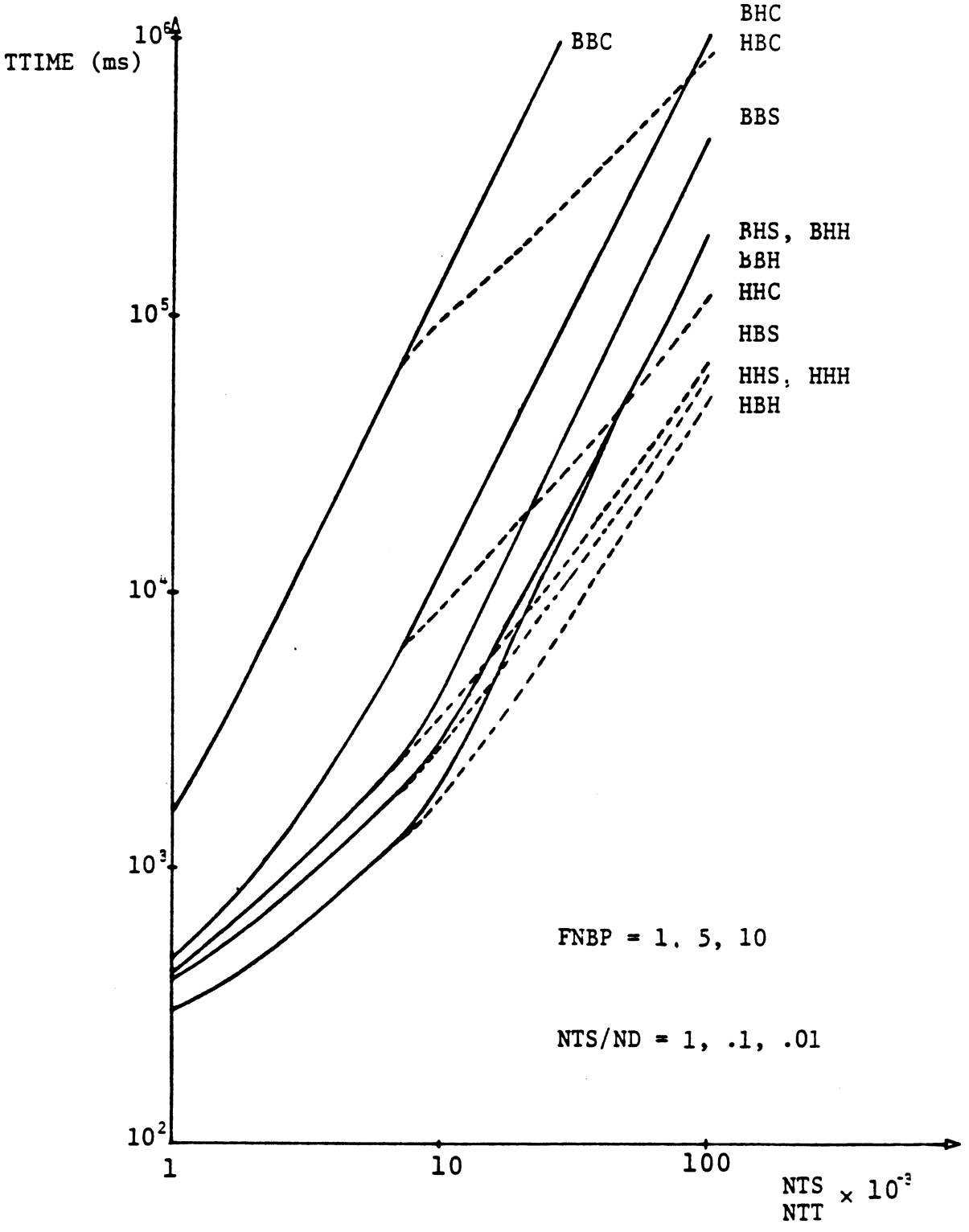

Figure 4.5 The Basic Equi-Join Algorithms

Basic algorithms is superior to that of the BBC-Basic algorithms. Therefore, the former algorithm is recommended for performing these operations on the proposed RDBM

4.3.2. The Evaluation of the TPF Equi-Join Algorithms

A computer program was written to compute the performance measure TTIME for every TPF equi-join algorithm using the equations already developed in Section 4.2.2. The set of programs were run for NTS(NTT) ε $\{10^3 - 10^5\}$, [NTS/ND] ε {1, .1, .01}, FNBP = 1 and FNBIT ε {.1, 1, 10}. The parameter FNBP was restricted to the value of one since larger values do not affect TTIME, as was shown in the previous section.

The behavior of the performance measure TTIME which corresponds to the above parameters' values is shown in Figures 4.6 through 4.10. From these figures, the following important conclusions can be drawn:

1. For large values of the joining probability [NTS/ND ~ 1], the TTIME is independent of the parameter FNBIT (Figure 4.6). This is due to the fact that with [NTS/ND ~ 1], the vector, regardless of its size, will filter out few target tuples, since most of these tuples would find a match among those of the source ones.

2. For large values of the joining probability, the best performing algorithm within the TPF category is HBS-TPF. To reach this conclusion reasoning similar to that used in conclusion Number 4 of the previous section was used.

3. For moderate values of the joining probability [NTS/ND ~ .1], the TTIME is generally dependent on the value of the parameter FNBIT (Figures 4.7 through 4.9). As FNBIT increases (NBIT gets larger) the performance of every TPF algorithm, over the range of NTS(NTT), is improved. However,

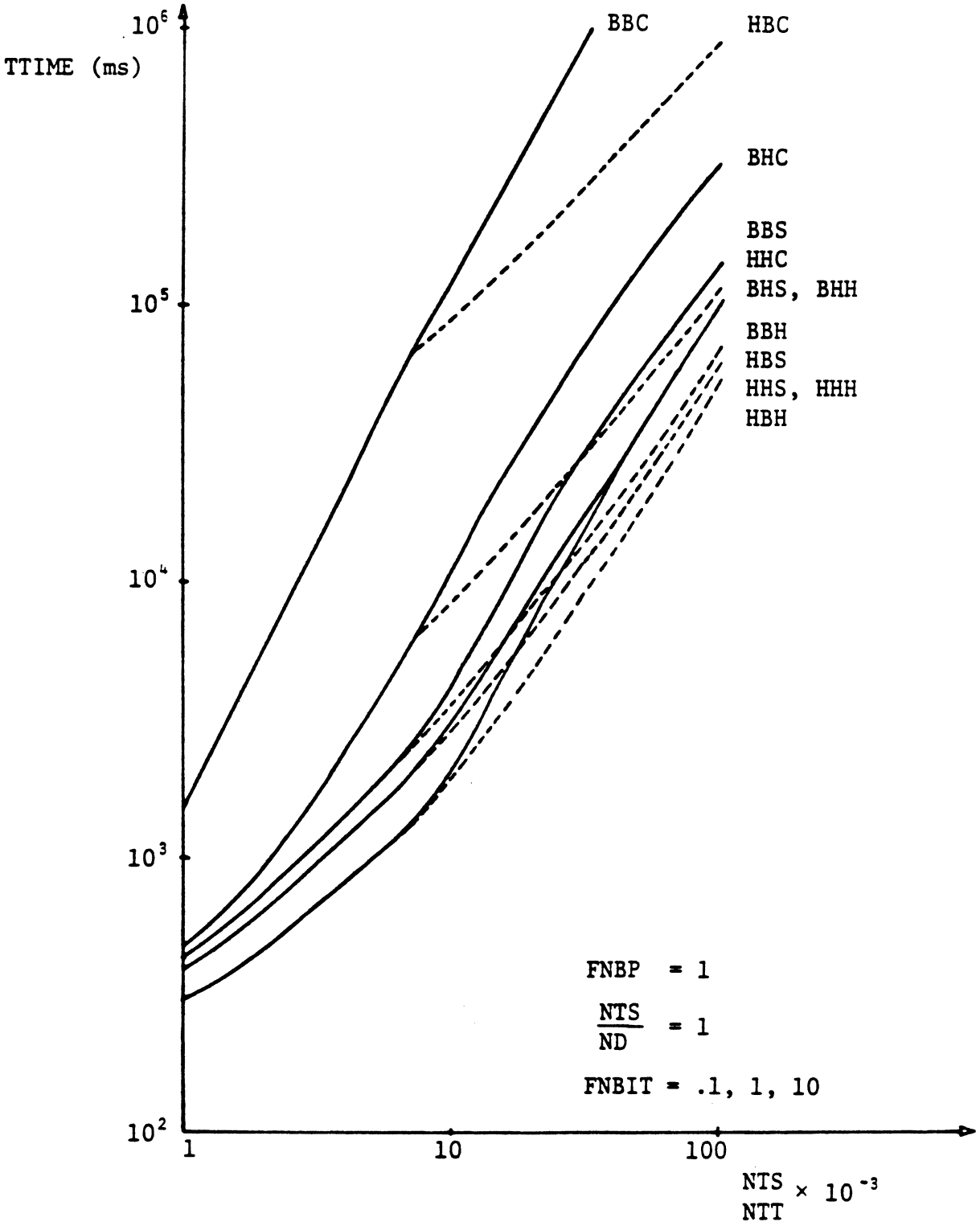

Figure 4.6 The Performance of the TPF Algorithms

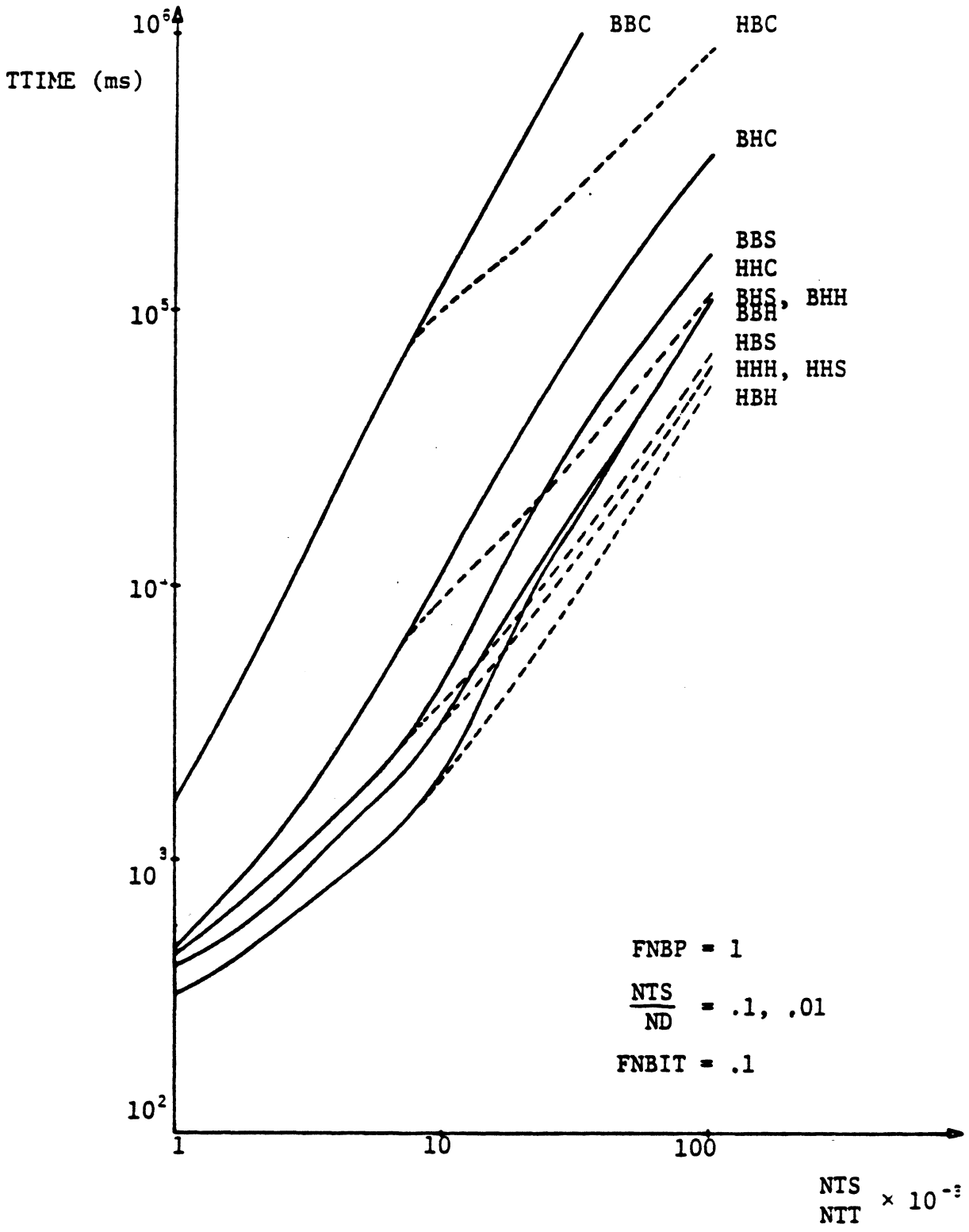

Figure 4.7 The Performance of the TPF Algorithms

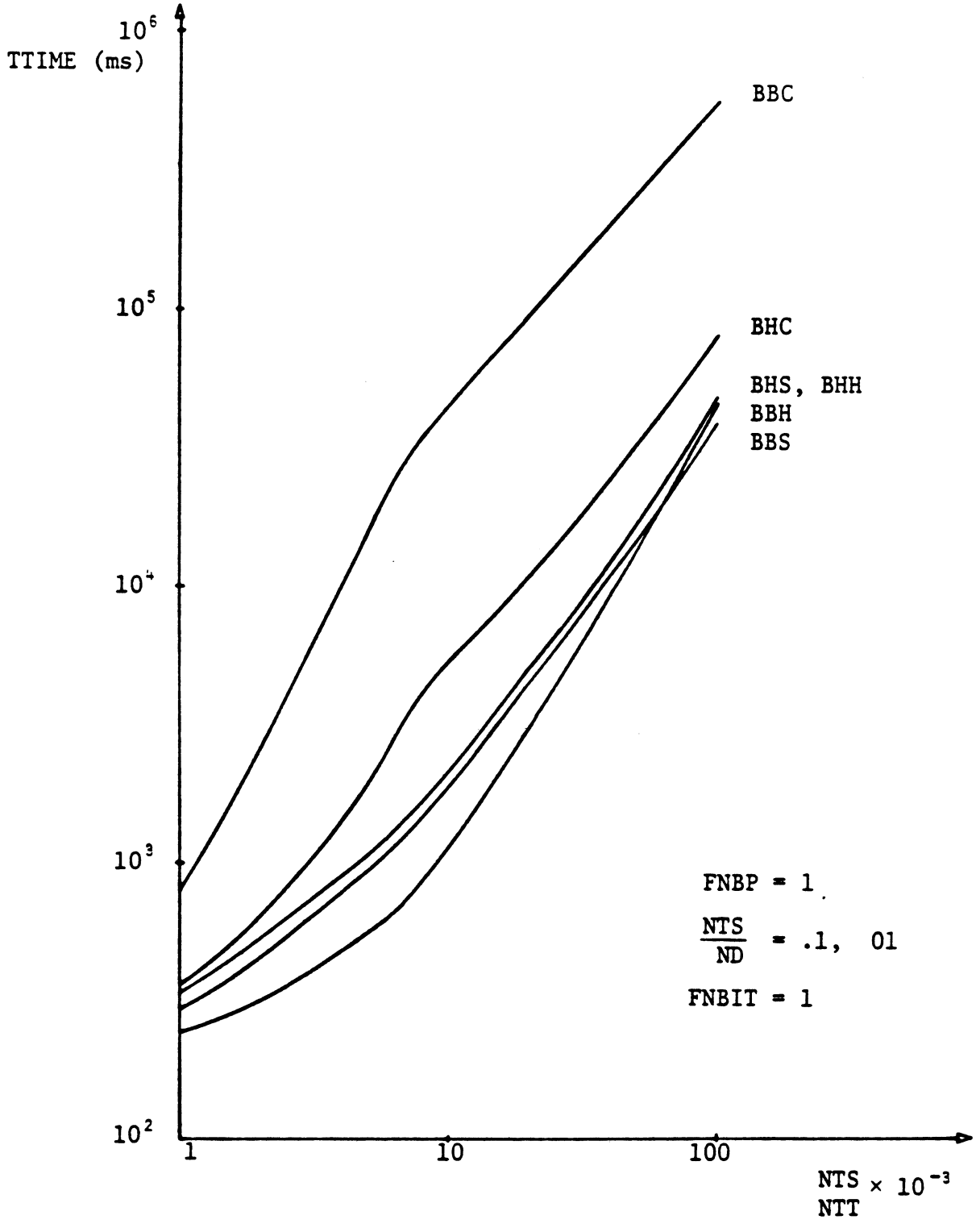

Figure 4.8 The Performance of the TPF Algorithms

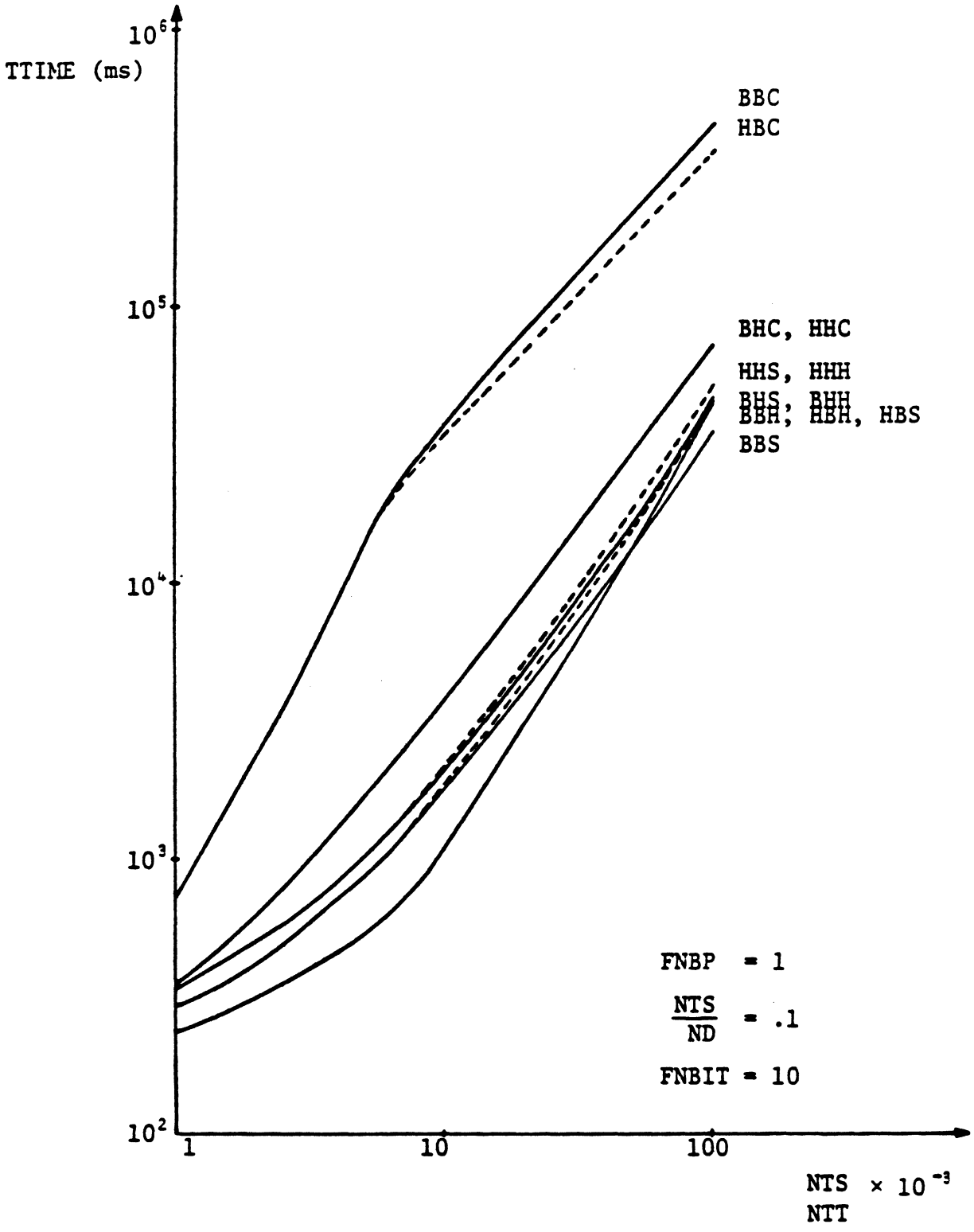

Figure 4.9 The Performance of the TPF Algorithms

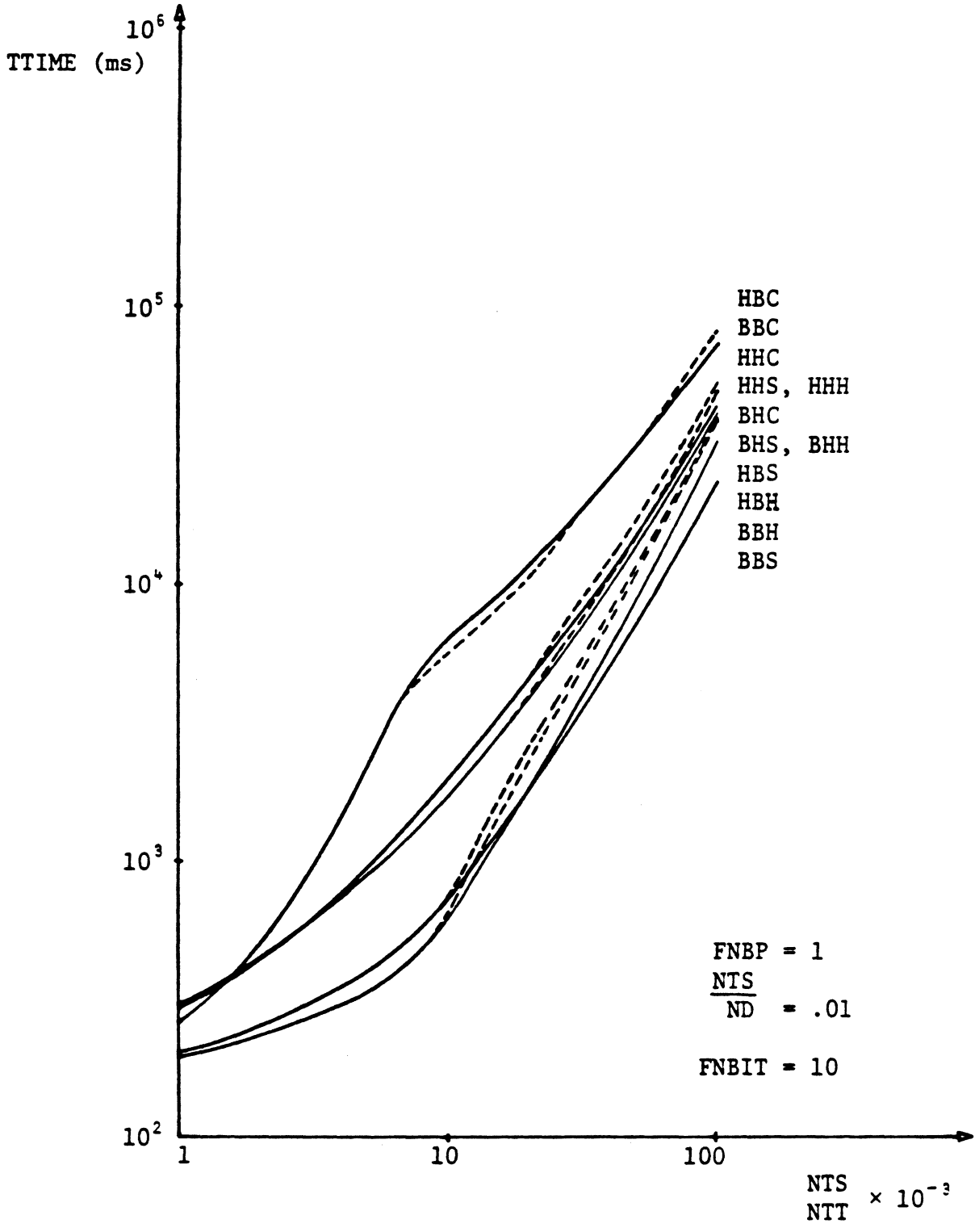

Figure 4.10 The Performance of the TPF Algorithms

this improvement becomes insignificant as FNBIT increases beyond the value of one (compare Figure 4.8 with Figure 4.9). Also the magnitude of the performance improvement due to the increase in the value of the parameter FNBIT is not the same for all the TPF algorithms. For example, the improvement in the performance of BBC-TPF algorithm is much larger than that of the HBH-TPF one.

4. For moderate values of the joining probability, the best performing algorithm within the TPF category is a function of FNBIT. For small FNBIT (~ .1), the HBS-TPF algorithm is the best performing one. On the other hand, the BBS-TPF algorithm is the best performing one for moderate to large values of FNBIT.

5. For small values of the joining probability (NTS/ND ~ .01), the TTIME is also generally dependent on the value of the parameter FNBIT (Figures 4.7, 4.8 and 4.10). However, this improvement continues as the value of FNBIT increases beyond the value of one.

6. For small values of the joining probability, the best performing algorithms are the same as those for the moderate joining probability case.

4.3.3. The evaluation of the STPF Equi-Join Algorithms

A computer program was written to compute the performance measure TTIME for every STPF equi-join algorithm using the equations already developed in Section 4.2.3. The set of programs were run for NTS(NTT) ε $\{10^3 - 10^5\}$, [NTS/ND] ε {1, .1, .01}, FNBP = 1 and FNBIT ε {.1, 1, 10}. The corresponding behavior of TTIME is shown in Figures 4.11 through 4.15. From these figures, the following important conclusions can be drawn:

1. For large values of the joining probability (NTS/ND ~ 1), the TTIME is independent of the parameter FNBIT (Figure 4.13). The reason for that is

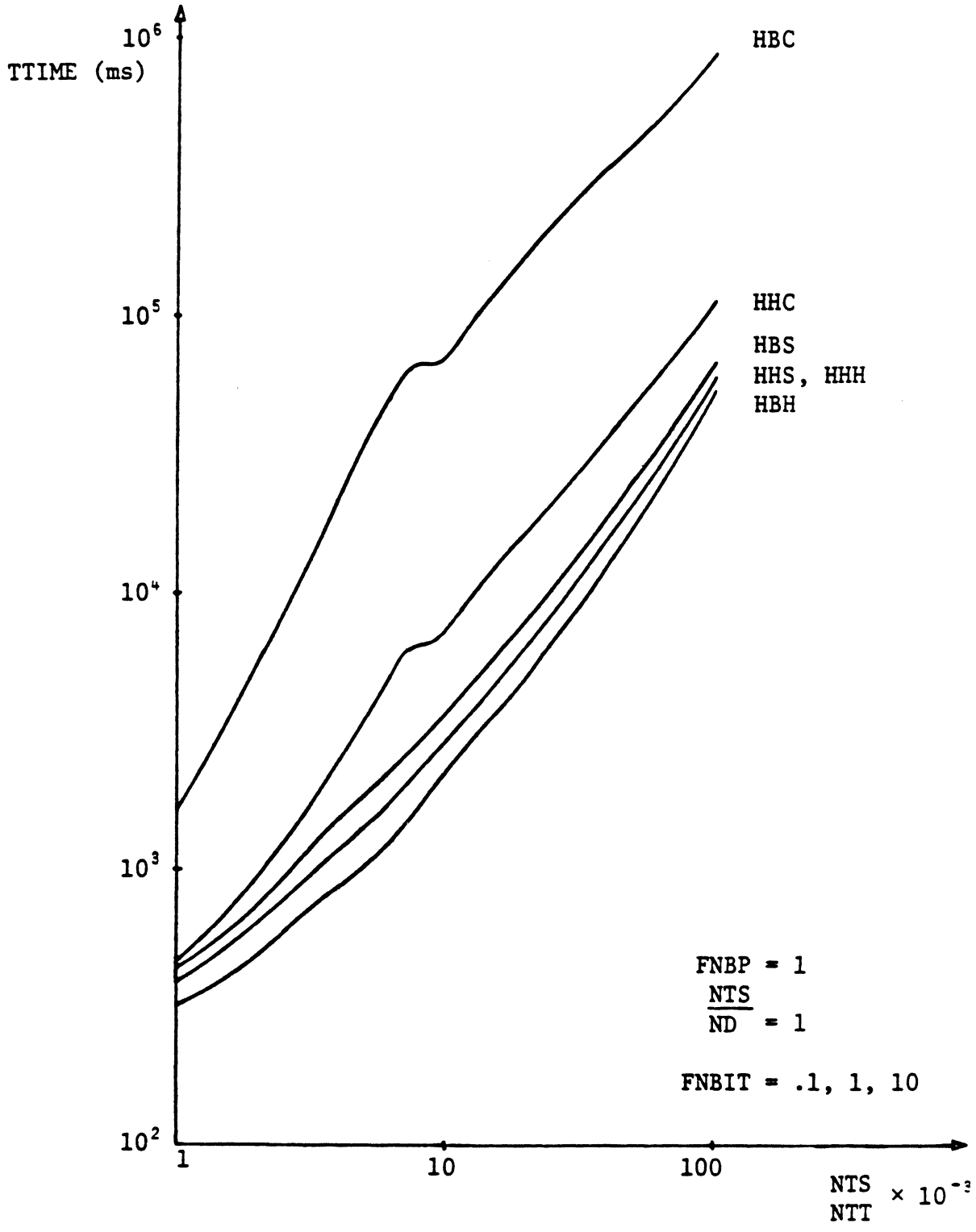

Figure 4.11 The Performance of the STPF Algorithms

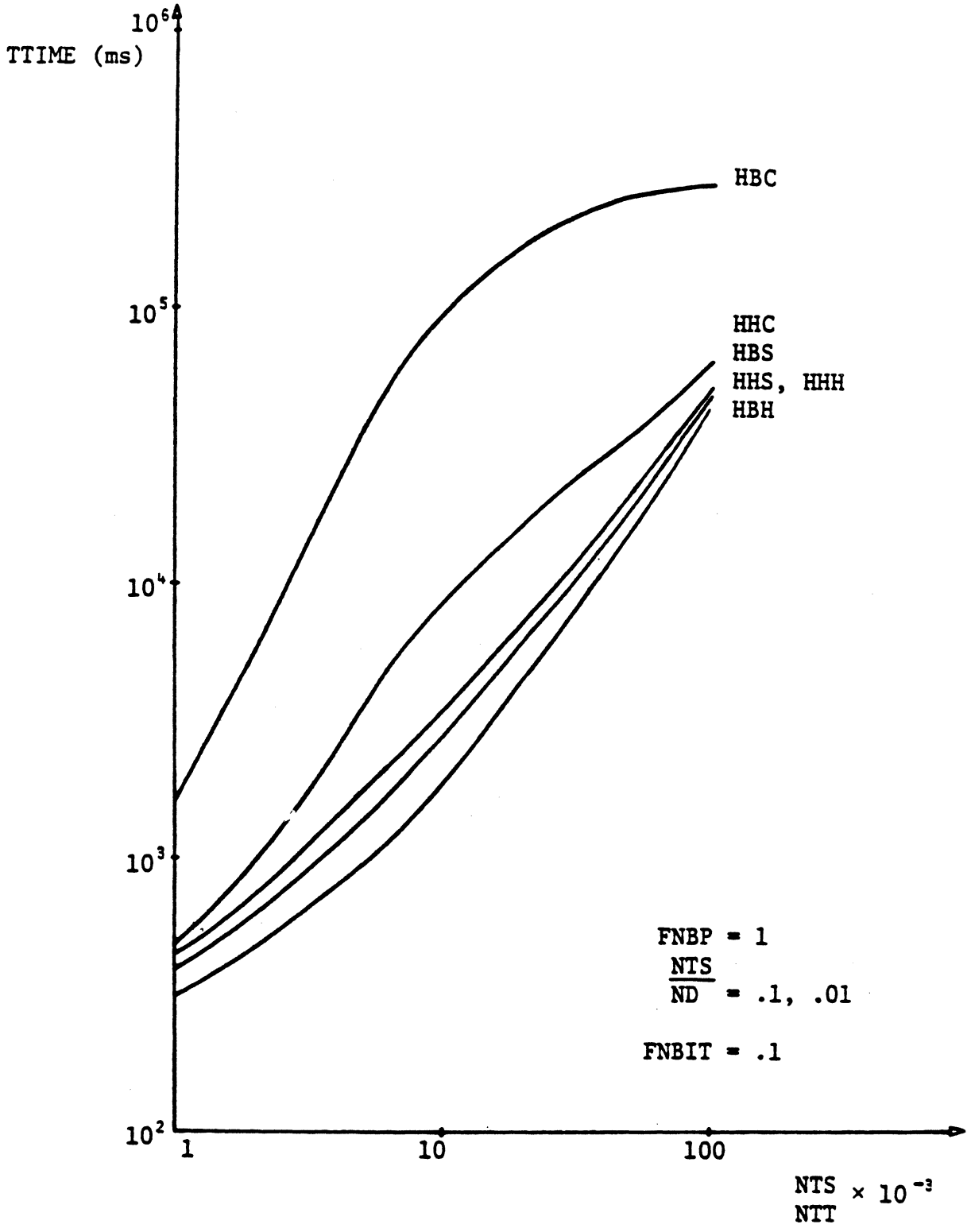

Figure 4.12 The Performance of the STPF Algorithms

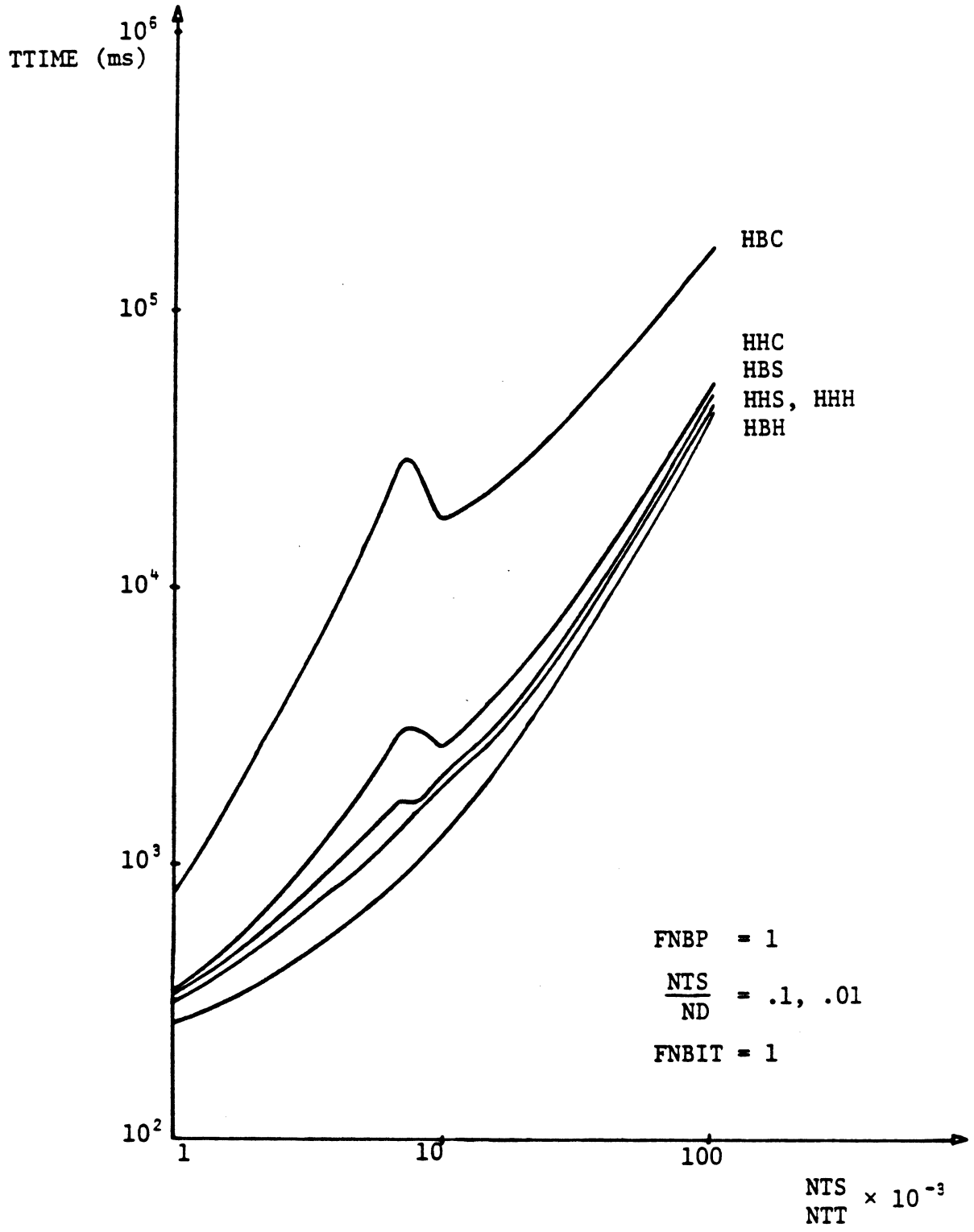

Figure 4.13 The Performance of the STPF Algorithms

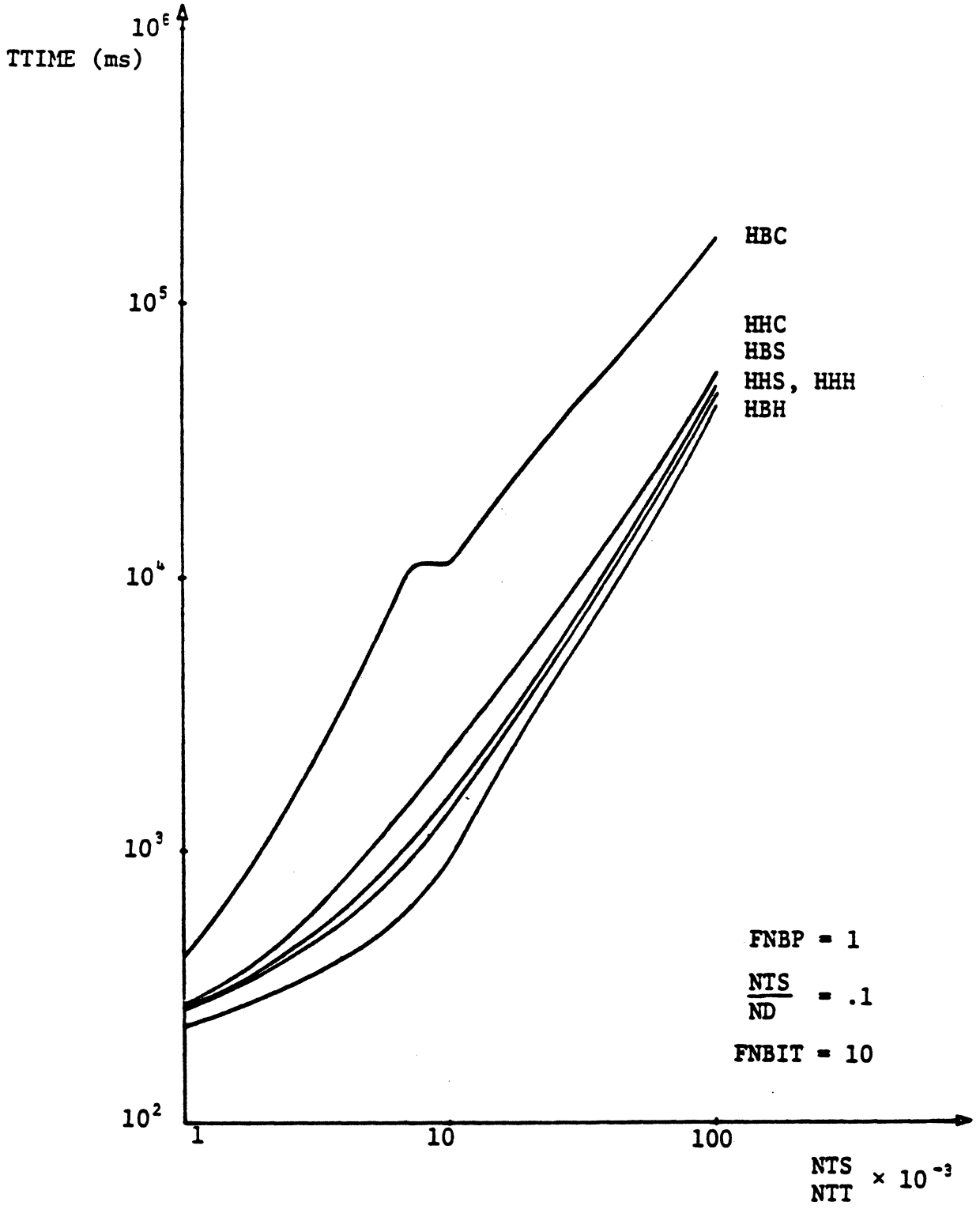

Figure 4.14 The performance of the STPF Algorithms

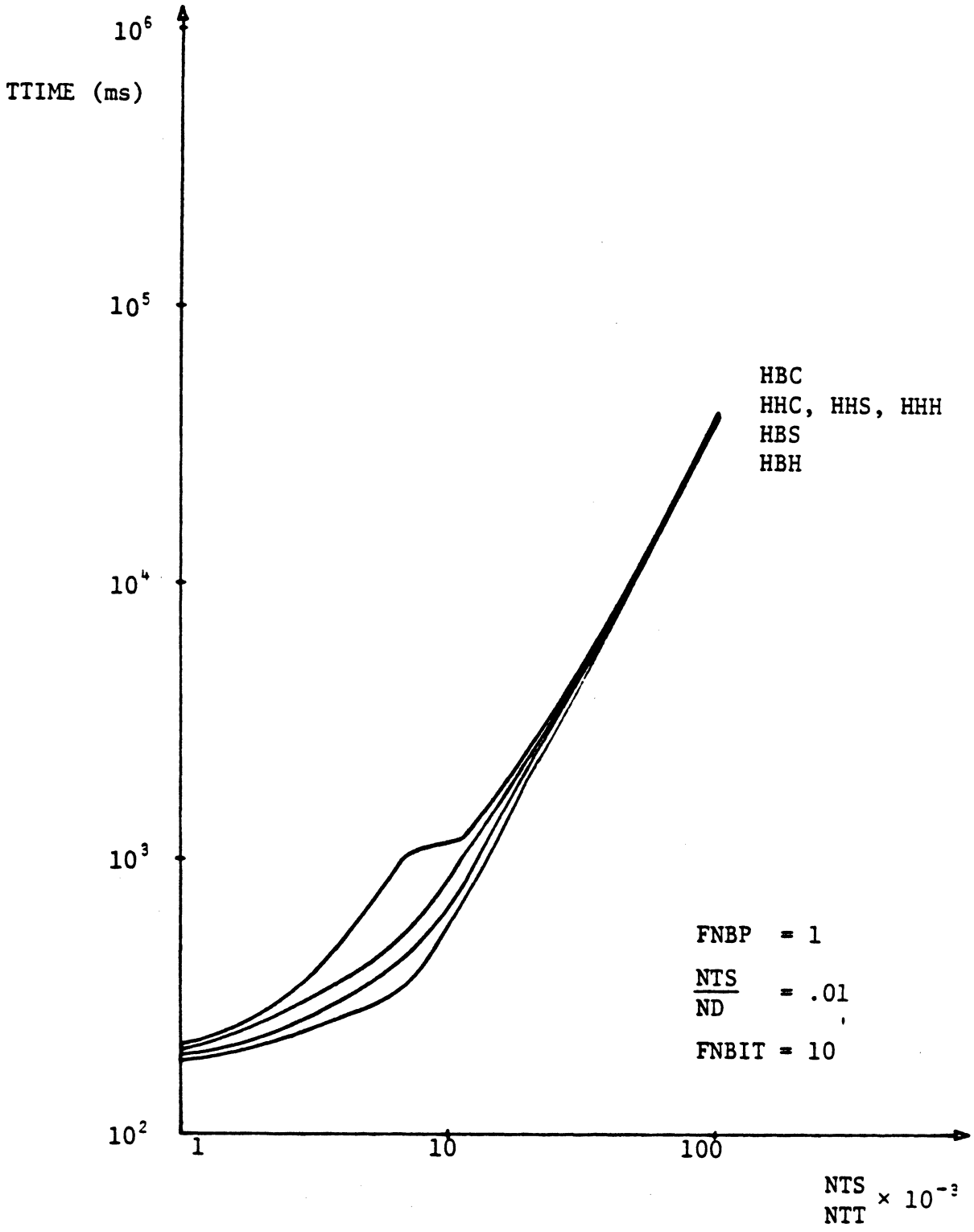

Figure 4.15 The Performance of the STPF Algorithms

the same as the one given in conclusion Number 1 of Section 4.3.2. It is also evident, from Figure 4.13, that for large values of the joining probability, the best performing algorithm within the STPF category is HBS-STPF.

2. For moderate values of the joining probability (NTS/ND ~ .1), the TTIME is generally dependent on the value of the parameter FNBIT (Figures 4.12 through 4.14). As FNBIT increases from .1 to 10, the performance of every STPF algorithm is improved. However, this improvement is insignificant for the STPF algorithms HBS, HBH, HHS and HHH, especially when used for joining large relations (NTS=NTT >60000 tuples). It is also evident, from Figures 4.12 through 4.14, that for all values of FNBIT and moderate values of the joining probability, the best performing algorithm within the STPF category is HBS-STPF.

3. For small values of the joining probability (NTS/ND ~ .01), the TTIME is also dependent on the value of the parameter FNBIT (Figures 4.12, 4.13 and 4.15). From the latter figures it is evident that for all values of FNBIT and small values of the joining probability the best performing algorithm within the STPF category is HBS-STPF.

4.3.4. The Evaluation of the STCF Equi-Join Algorithms

A computer program was written to compute the performance measure TTIME for those algorithms of the STCF category which were modeled in Section 4.2.4. The set of programs were run for NTS(NTT) ε $\{10^3 - 10^5\}$, [NTS/ND] ε {1, .1, .01} and FNBIT ε {1, .1, .01}. The corresponding behavior of TTIME is shown in Figures 4.16 through 4.19. From these figures, the following important conclusions can be drawn:

1. For large values of the joining probability (NTS/ND ~ 1), the TTIME is independent of the parameter FNBIT (Figure 4.16). The best performing

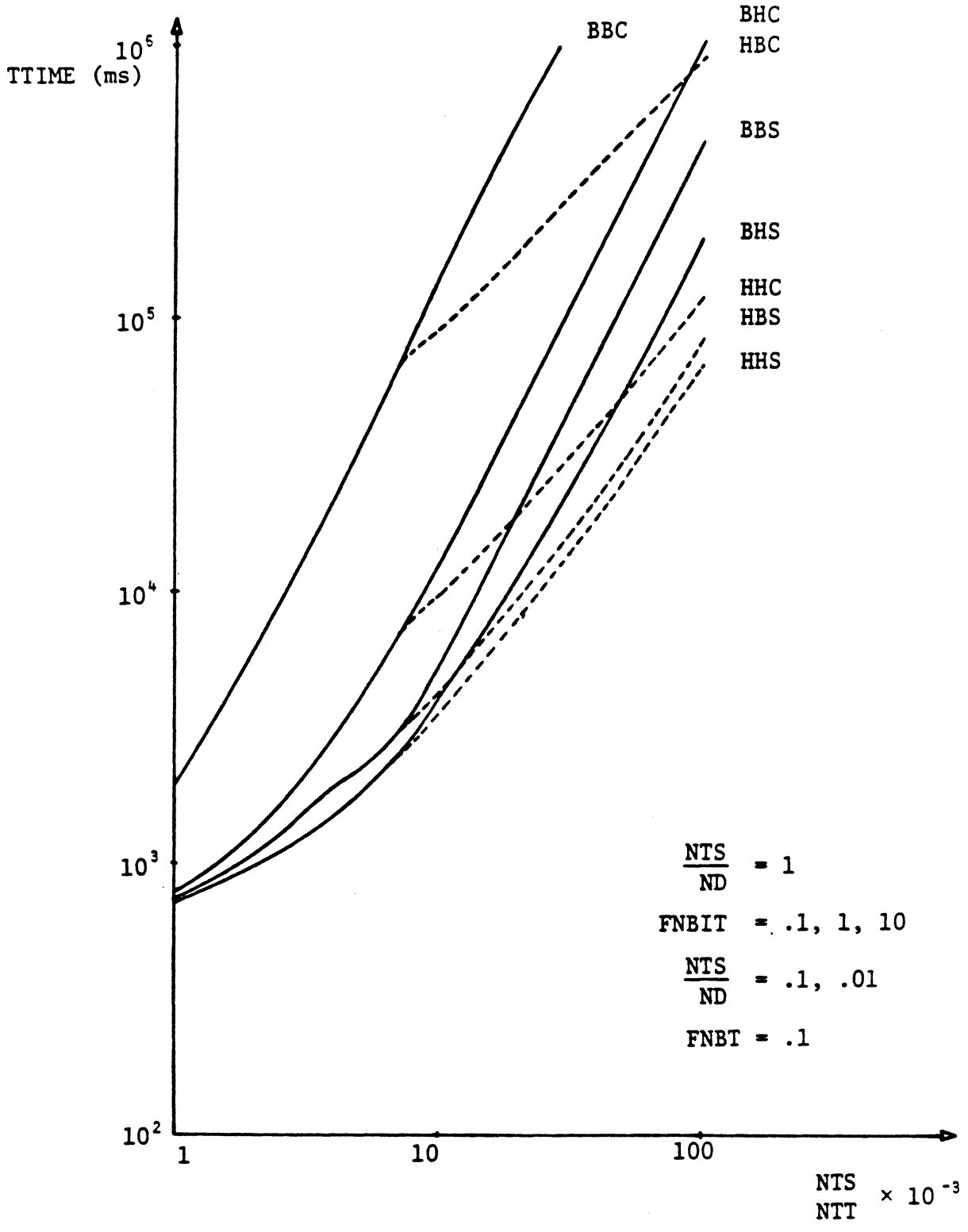

Figure 4.16 The Performance of the STCF Algorithms

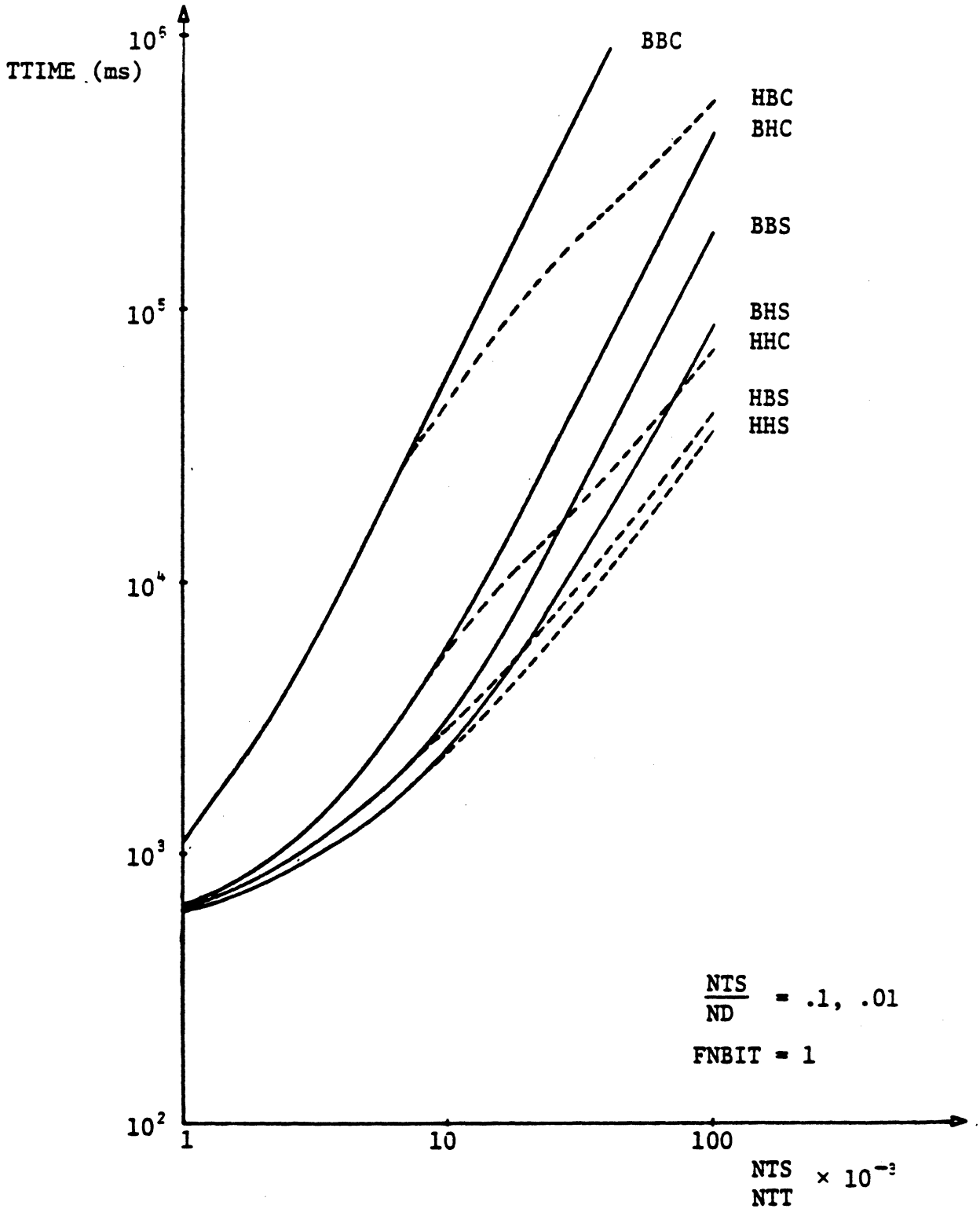

Figure 4.17 The Performance of the STCF Algorithms

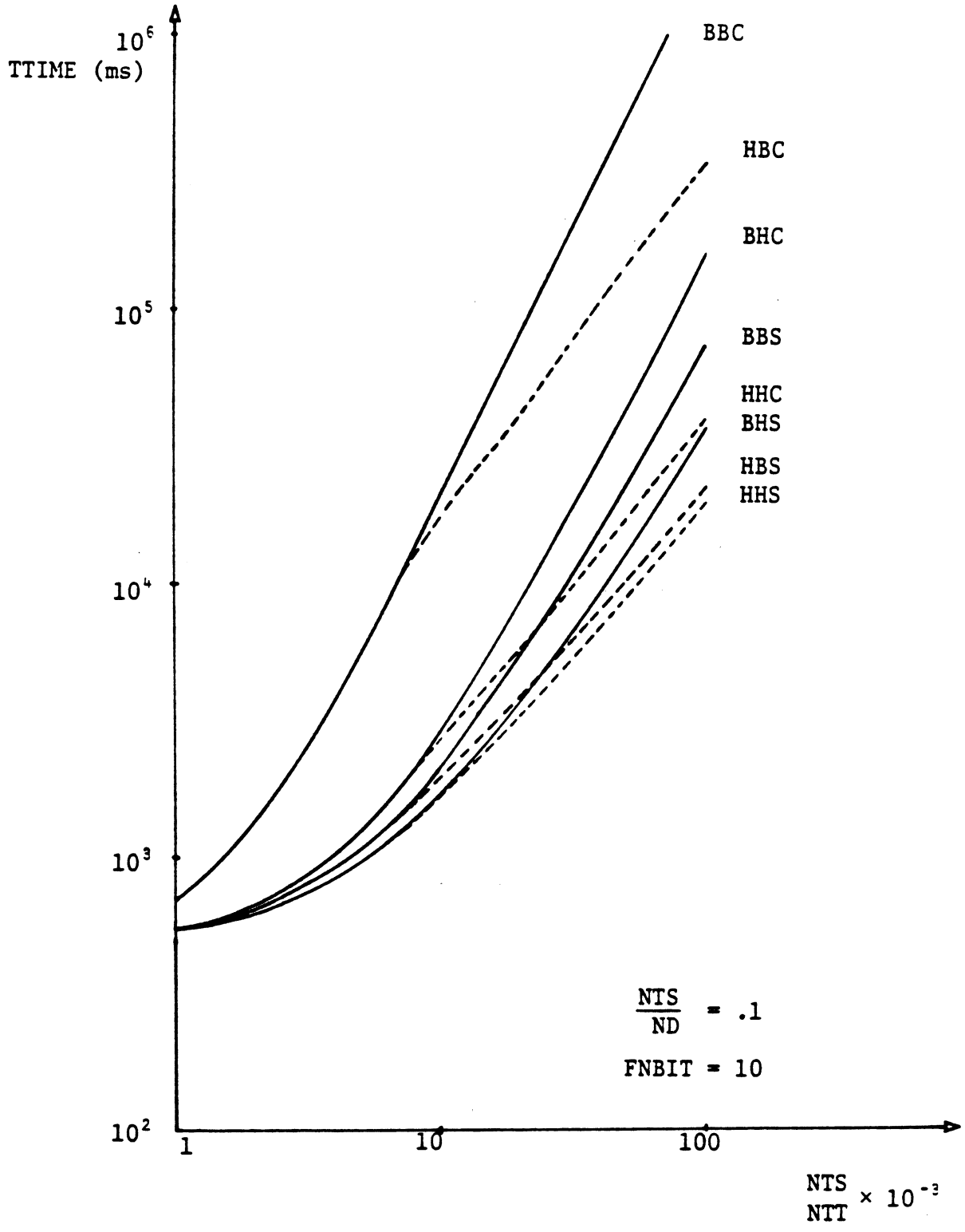

Figure 4.18 The Performance of the STCF Algorithms

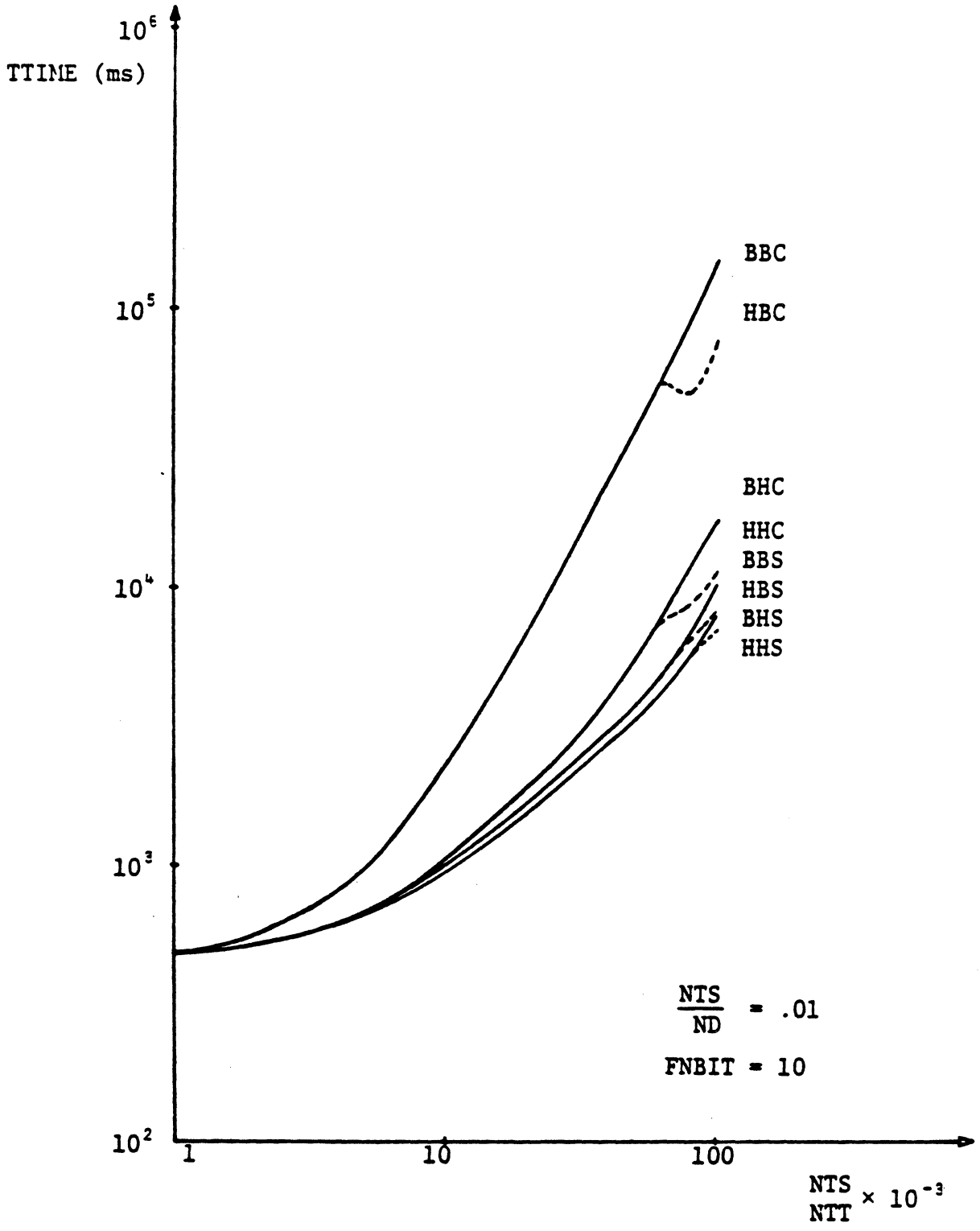

Figure 4.19 The performance of the STCF Algorithms

algorithm, in this data environment, within the STCF category is HBS-STCF.

2. For moderate values of the joining probability (NTS/ND ~ .1), the TTIME depends of the parameter FNBIT (Figures 4.16 through 4.19). Within this data environment, the best performing algorithm within the STCF category is HBS-STCF.

3. For small values of the joining probability (NTS/ND ~ .01), the TTIME is also depends on the parameter FNBIT (Figure 4.16, Figure 4.17 and Figure 4.19). The best performing algorithm within the STCF category is a function of the parameter FNBIT. For small FNBIT, the best performing algorithm is HBS-STCF. On the other hand, for large FNBIT the best performing one is BBS-STCF.

4.3.5. Comparing the Best Performing Equi-Join algorithms

One of the most important objectives for the performance evaluation, carried out earlier, is to determine the best performing equi-join algorithms for the various input data environments. In Sections 4.3.1 through 4.3.4 the best performing equi-join algorithms within the basic, TPF, STPF and STCF algorithmic categories were determined. Which algorithm within these categories is "the best performing one" was found to be a function of the parameters NTS/ND and FNBIT. However, when FNBIT assumes large values (>1), "which algorithm within each category is the best performing one" becomes a function of only one parameter, namely, [NTS/ND]. In this section, the best performing algorithms for the various algorithmic categories are compared. This comparison is carried out under the assumption that FNBIT=10. This comparison will enable us to determine the best performing equi-join algorithm(s) under the various input data environment.

In Figures 4.20 through 4.22 the performance measure TTIME is plotted for the best performing algorithms of the different algorithmic categories, verses the parameter NTS(NTT). The three figures correspond to the values 1, .1 and .01 of the parameter [NTS/ND]. In all of these figures, the parameter FNBIT is held constant at the value of one. From these figures, the following important conclusions can be drawn:

1. For large values of the joining probability (NTS/ND ~ 1), the algorithm HBS-basic is the overall best performing one (see Figures 4.20). Any other algorithm will either have an inferior performance (HBS-STCF) or equal performance but will require more overhead and storage (HBS-TPF and HBS-STPF). The performance of the HBS-Basic represents the upper bound on TTIME spent by the RDBM in executing the equi-join operation.

2. For moderate values of the joining probability (NTS/ND ~ .1), the overall best performing algorithm, in executing the equi-join operation, is the BBS-TPF algorithm (see Figure 4.21). Any other algorithm will either have an inferior performance (HBS-Basic, HBS-STPF when joining relations with large cardinalities and the HBS-STCF when joining relations with small cardinalities) or little performance advantage that does not justify the extra overhead these algorithms involve (HBS-STPF when joining relations with small cardinalities and HBS-STCF when joining relations with large cardinalities).

3. For small values of the joining probability (NTS/ND ~ .01), the overall best performing algorithm in executing the equi-join operation is a function of the number of tuples participating in the operation (refer to Figure 4.22). For relations with small cardinalities [NTS(NTT) < 15000 tuples], the best performing algorithm is the BBS-TPF. On the other hand, the best performing algorithm for relations with large cardinalities

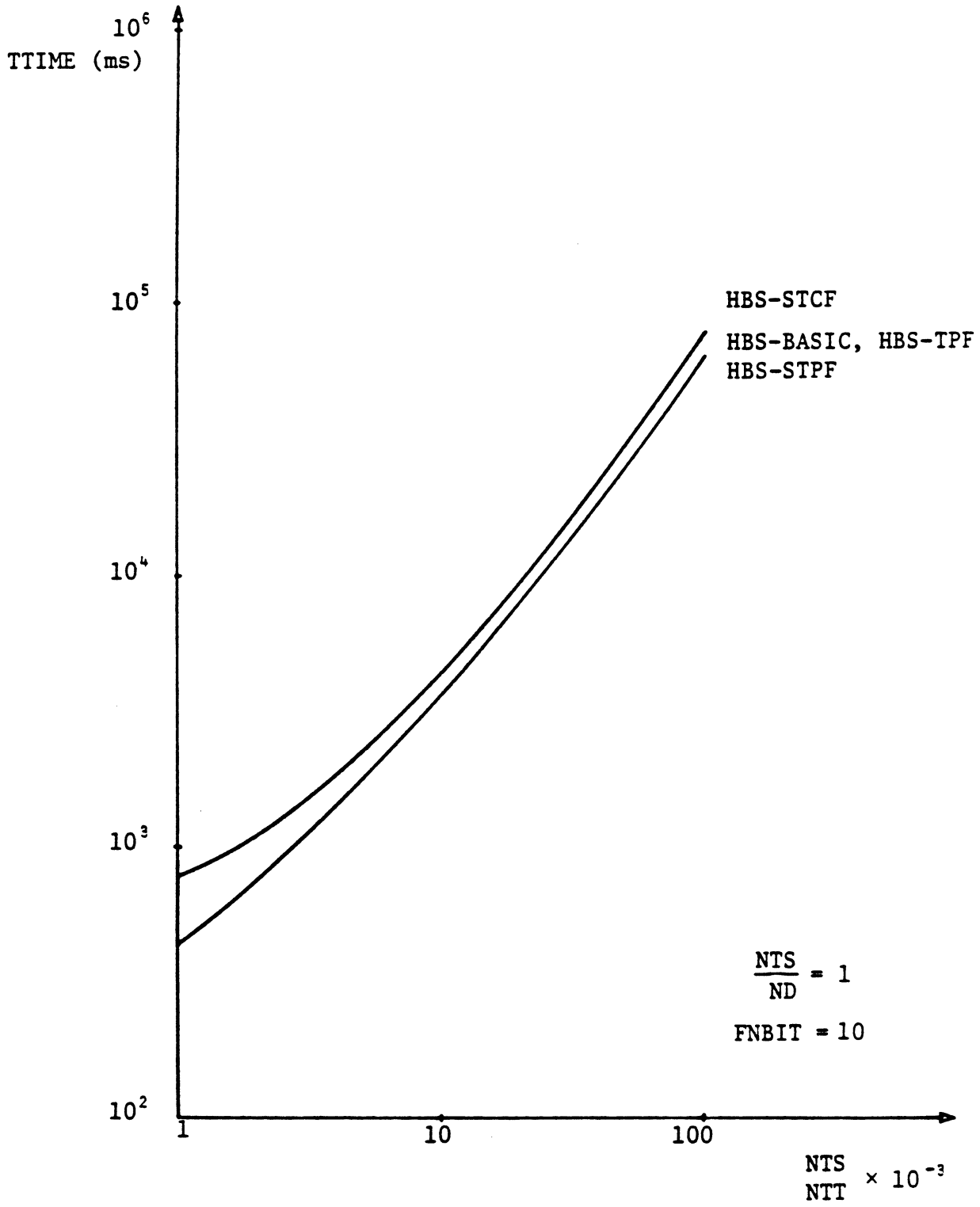

Figure 4.20 The Best Performing Equi-Join Algorithm Within Each Algorithmic Category

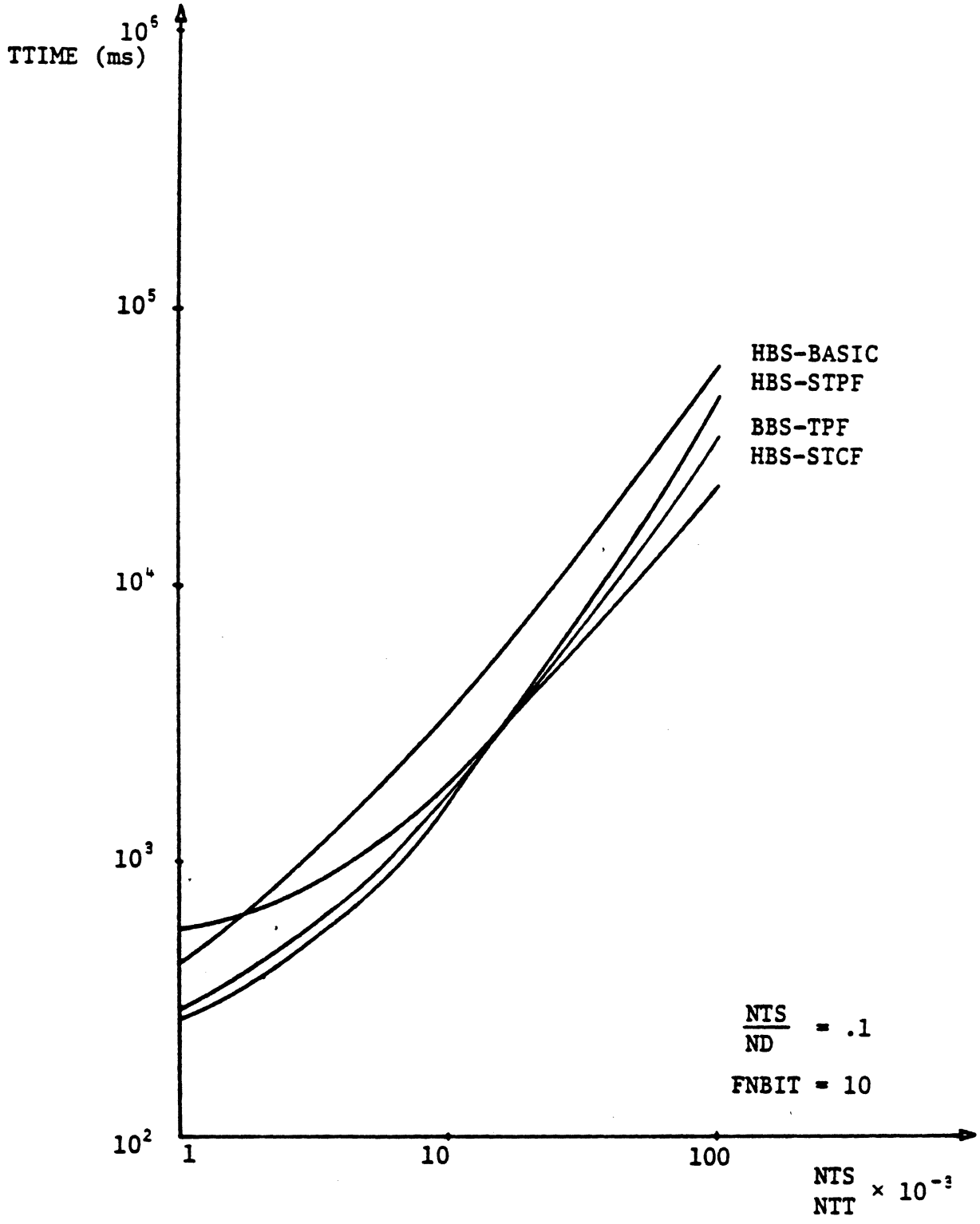

Figure 4.21 The Best Performing Equi-Join Algorithm Within Each Algorithmic Category

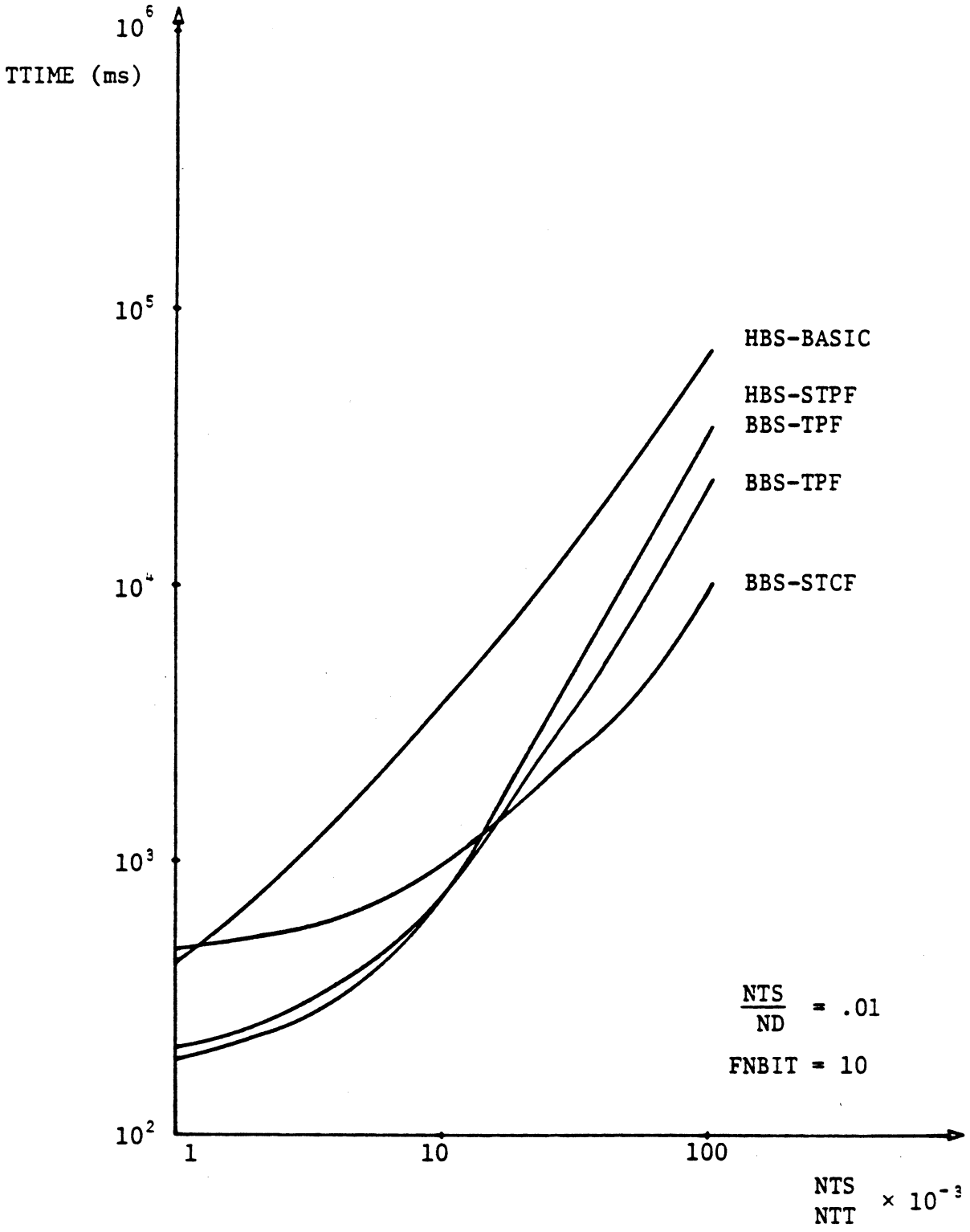

Figure 4.22 The Best Performing Equi-Join Algorithms Within Each Algorithmic Category

(> 15000 tuples) is the BBS-STCF.

Based on the preceding conclusions, it is recommended that the algorithms shown in Figure 4.23 are adopted for the new RDBM.

4.4. The Effect of Improving the Cluster's Intertriplets Communication on the Performance of the Equi-Join Operation

Recall from Chapter 3 that a processing cluster of the newly proposed RDBM is organized as a set of triplets which intercommunicate, indirectly, over a single bus, the TBUS. During the modeling process of Section 4.2, it was felt that the adoption of this bus structure would not allow some equi-join algorithms to capture their full performance potential. These algorithms are those which use the local hash method to distribute the tuples of both the source and the target relations among the triplets of a PC for processing. In general, these algorithms require more tuples to be moved among the cluster's triplets than those algorithms which use the local broadcast method.

In this section, the effect, on the performance of the Equi-Join operation, of providing the PC with a communication structure with higher bandwidth than the single bus can provide is investigated . One of these structures is the multiple bus structure. That is, in addition to the TBUS, the PC is provided with one or more buses. Each of the added ones is similar in architecture, bandwidth and function to that of the TBUS.

The addition of one or more buses to the PC will not affect the performance of the equi-join algorithms which use the local broadcast method. On the other hand, it will affect the performance of the equi-join algorithms which use the local hash method. The performance measure TTIME for the latter algorithms can be calculated using a slightly modified version of the corresponding models already developed in Section 4.2. The modification to these models involves updating the equations which compute the quantity

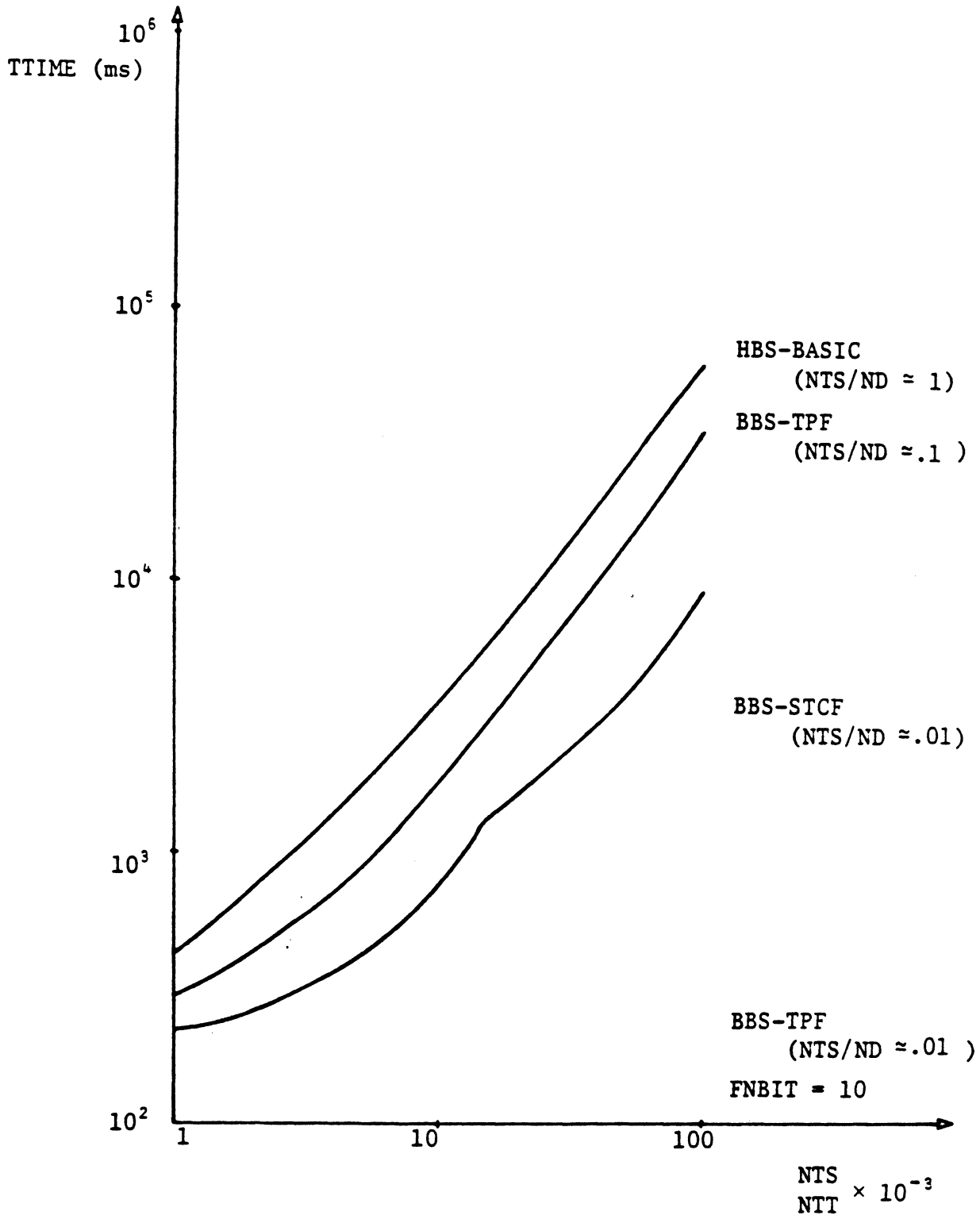

Figure 4.23 The Recommended Equi-Join Algorithms

TTB(1) to take into account the new multiple bus organization.

Let NBUS = the number of buses (including the TBUS) in a PC,

TTB(1) = the bus transmission time per phase for the single bus organization, and

TTB^* = the multiple bus transmission time per phase.

It is easy to see that

$$TTB^*(1) = \frac{TTB(1)}{NBUS}$$

where NBUS $\varepsilon \left\{1,2,\ldots \left\lfloor \frac{NP}{2} \right\rfloor\right\}$.

The equations of Section 4.2 which compute the TTIME for the "local hash" equi-join algorithms were updated according to the above equation. The corresponding computer programs were updated also. The updated programs were run for NTS(NTT) ε $\{10^3 - 10^5\}$, [NTS/ND] ε { 1, .1, .01}. For the local hash-basic, -TPF, and -STPF algorithms, the parameter FNBIT is held constant at the value of ten. In order to obtain an upper bound on the performance improvement, the parameter NBUS is assigned its highest value $\left\lfloor\left\lfloor \frac{NP}{2} \right\rfloor\right\rfloor$. The corresponding behavior of the TTIME is shown in Figures 4.24 through 4.33. From these figures it can be concluded easily that the best performing algorithm within each of the local hash-basic, -TPF, -STPF and -STCF categories is independent of the parameter [NTS/ND]. The best performing algorithms are: the BHS-Basic,the BHS-TPF, the HHS-STPF and the BHS-STCF for the local hash-basic, -TPF, -STPF and -STCF algorithmic categories, respectively.

In Figures 4.34 through 4.36 the TTIME of the best performing algorithms, determined above, verses NTS(NTT) are plotted for the [NTS/ND] ε {1, .1, .01},

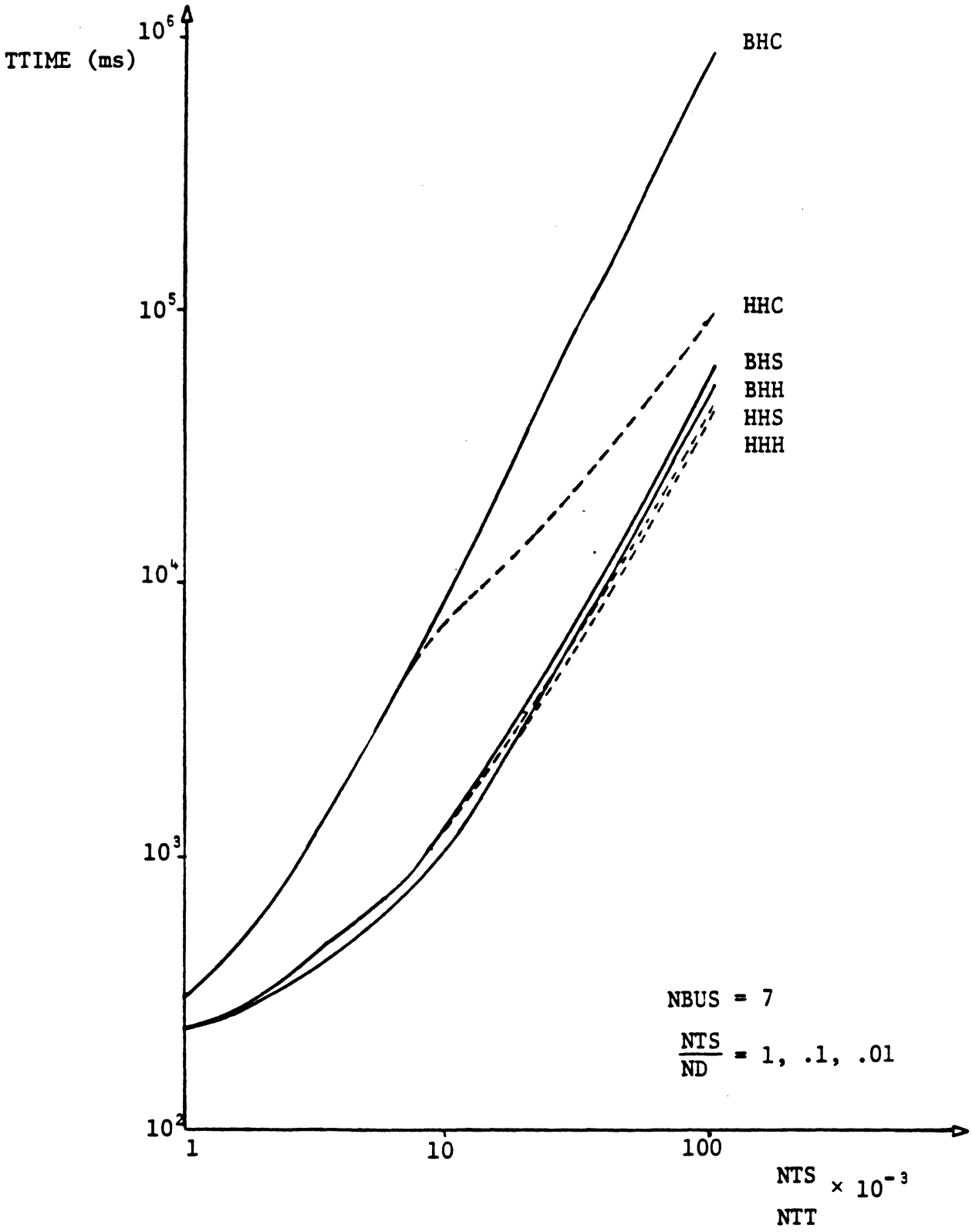

Figure 4.24 The Performance of the "Local Hash" Basic Algorithm

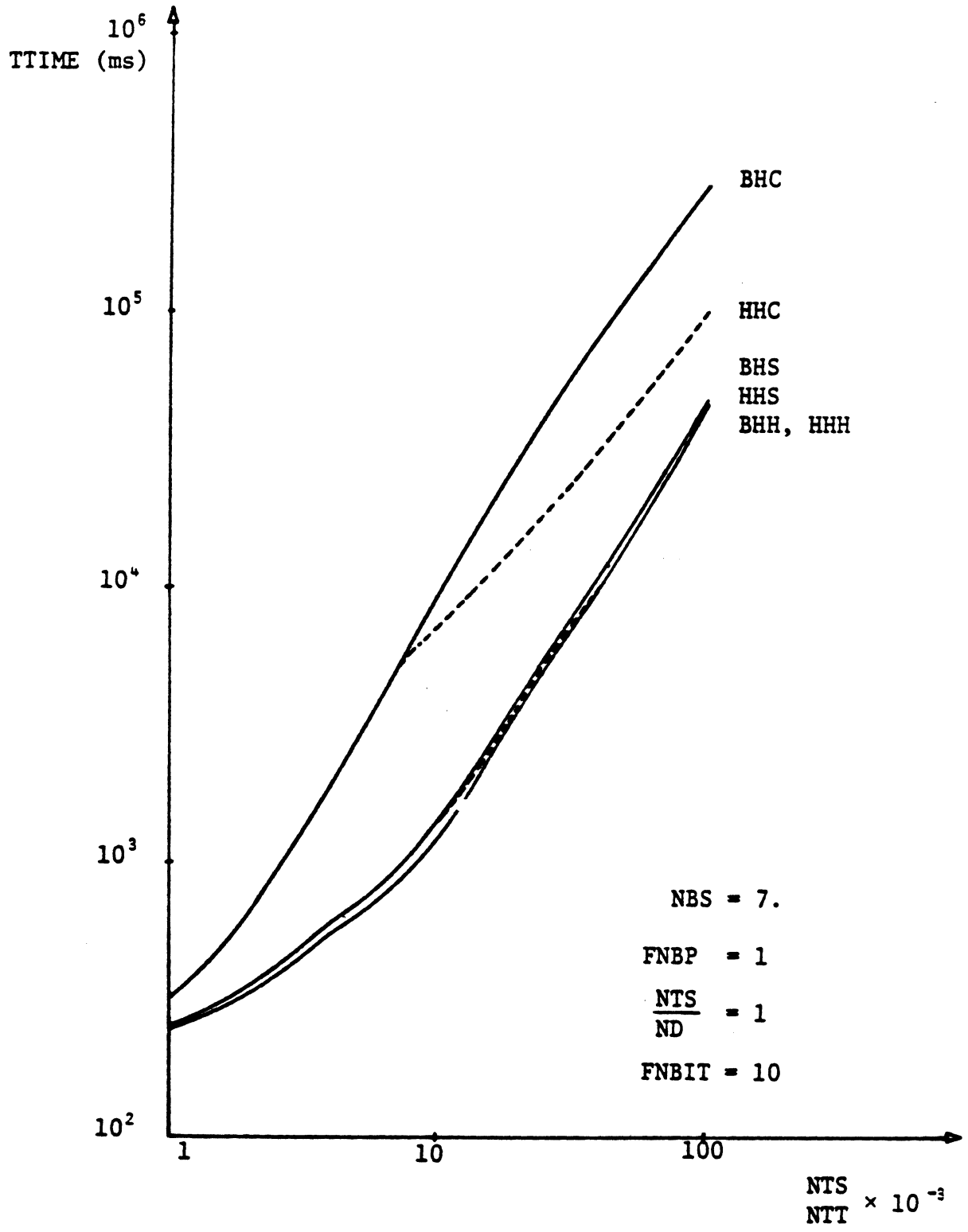

Figure 4.25 The Performance of the "Local Hash" TPF Algorithms

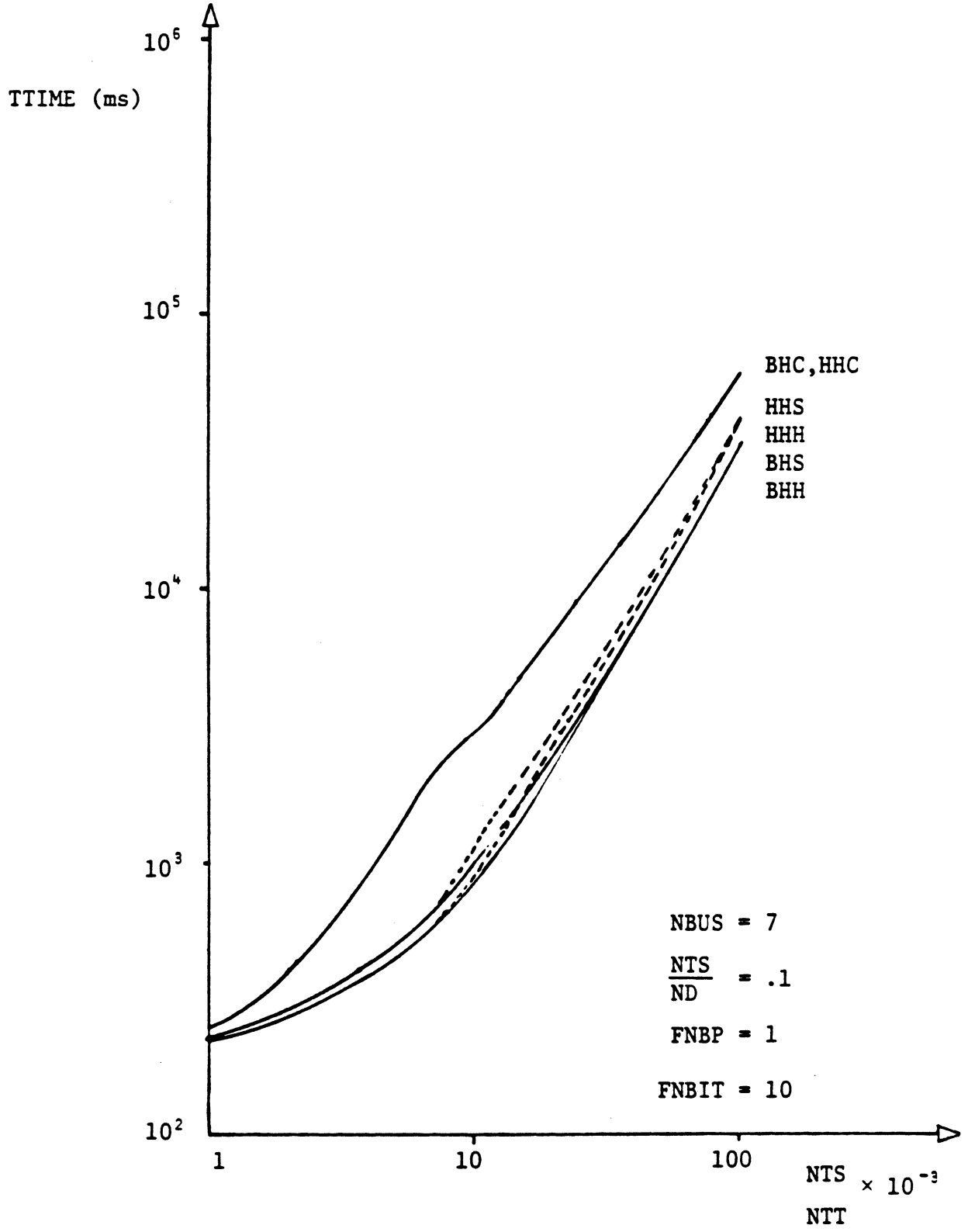

Figure 4.26 The Performance of the "Local Hash" TPF Algorithms

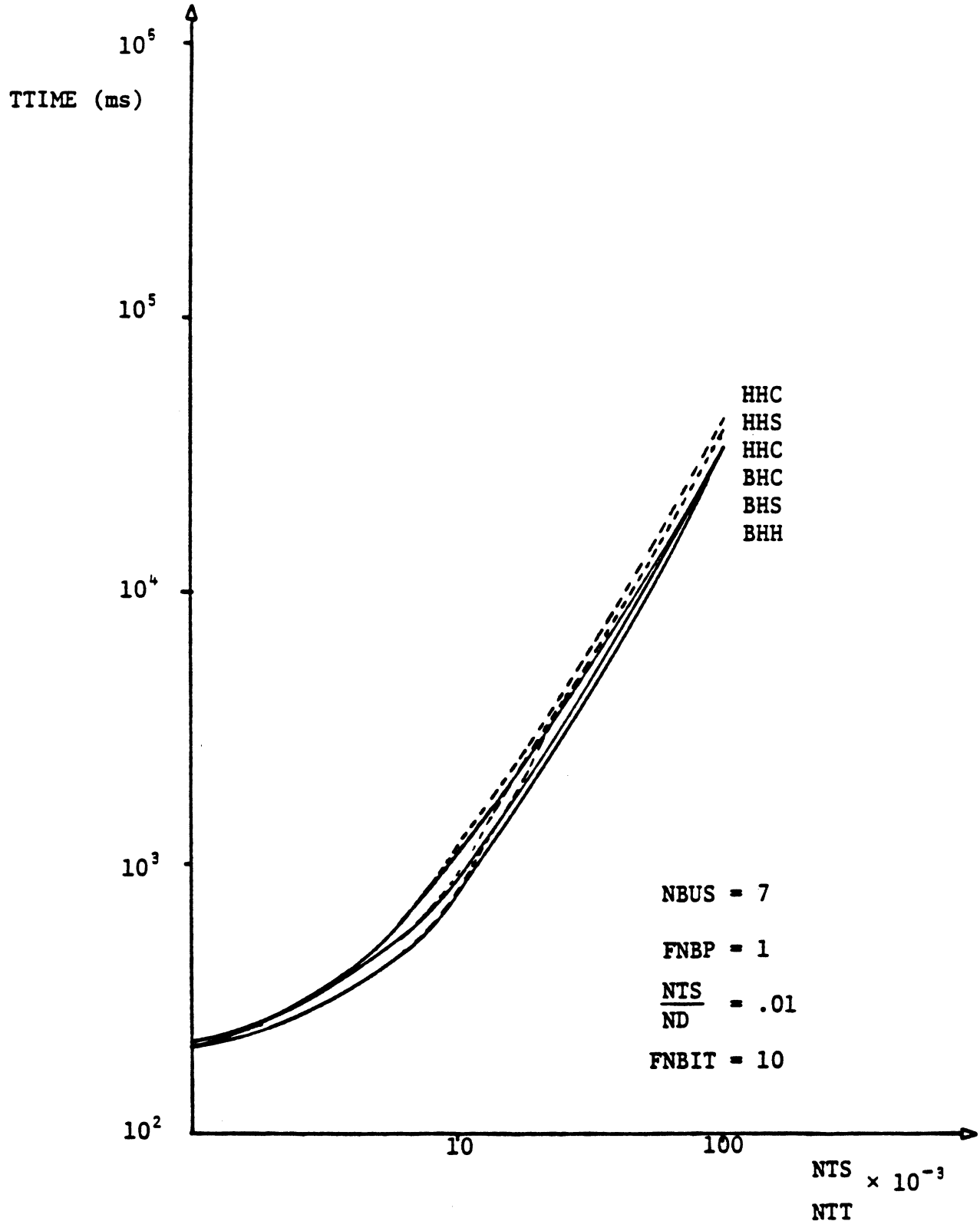

Figure 4.27 The Performance of the "Local Hash" TPF Algorithms

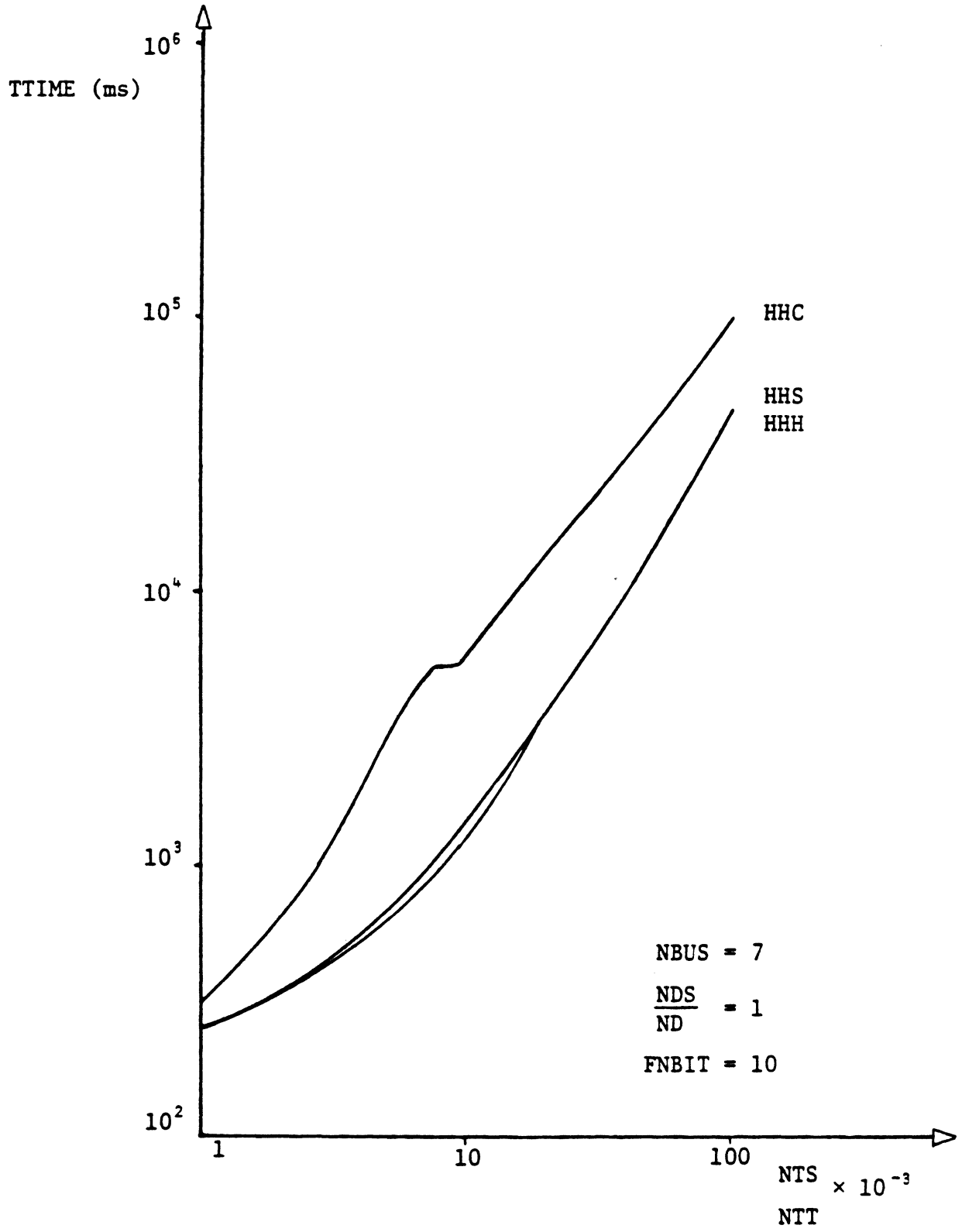

Figure 4.28 The Performance of the "Local Hash"

STPF Algorithms

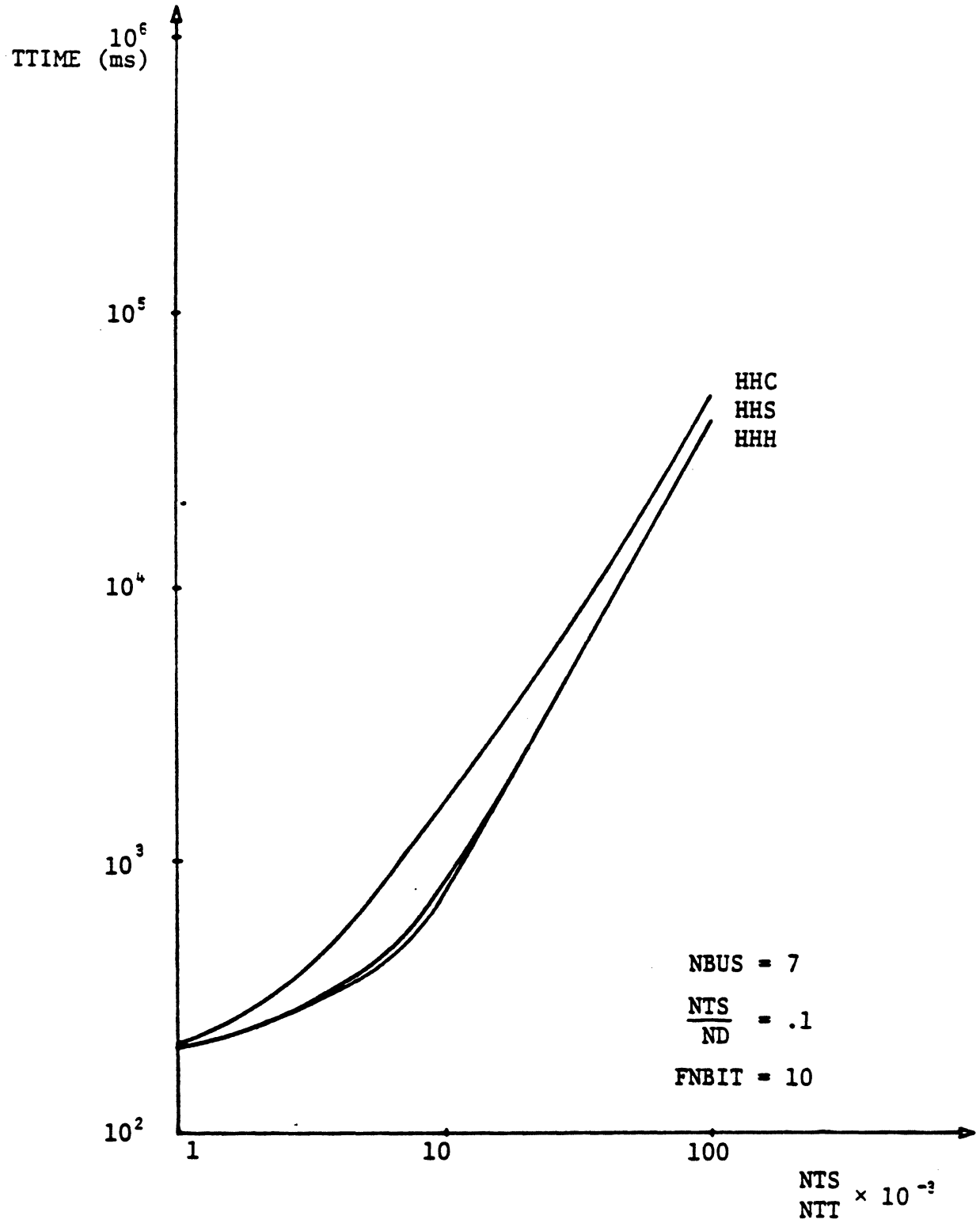

Figure 4.29 The Performance of the "Local Hash" STPF Algorithms

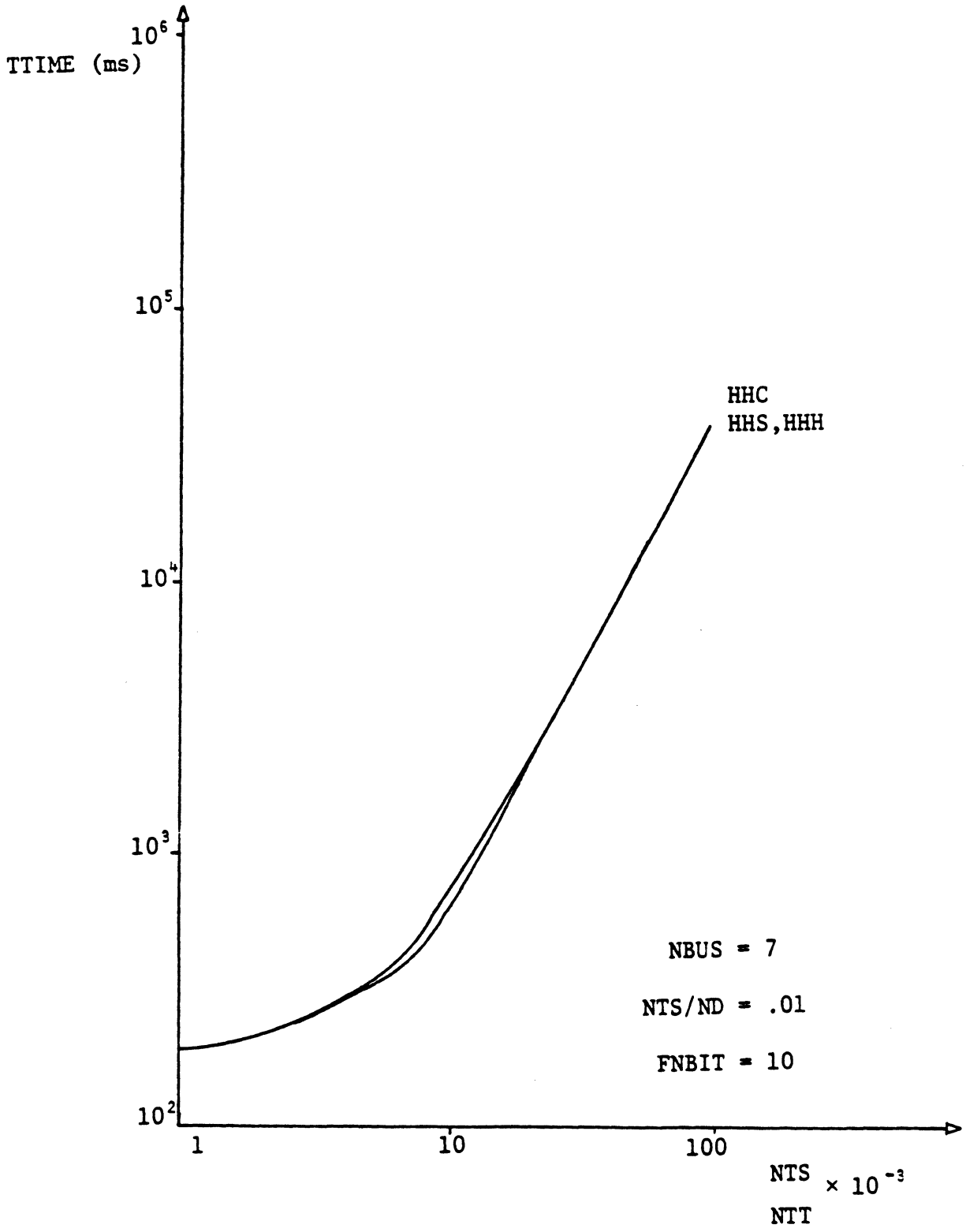

Figure 4.30 The Performance of the "Local Hash" STPF Algorithms

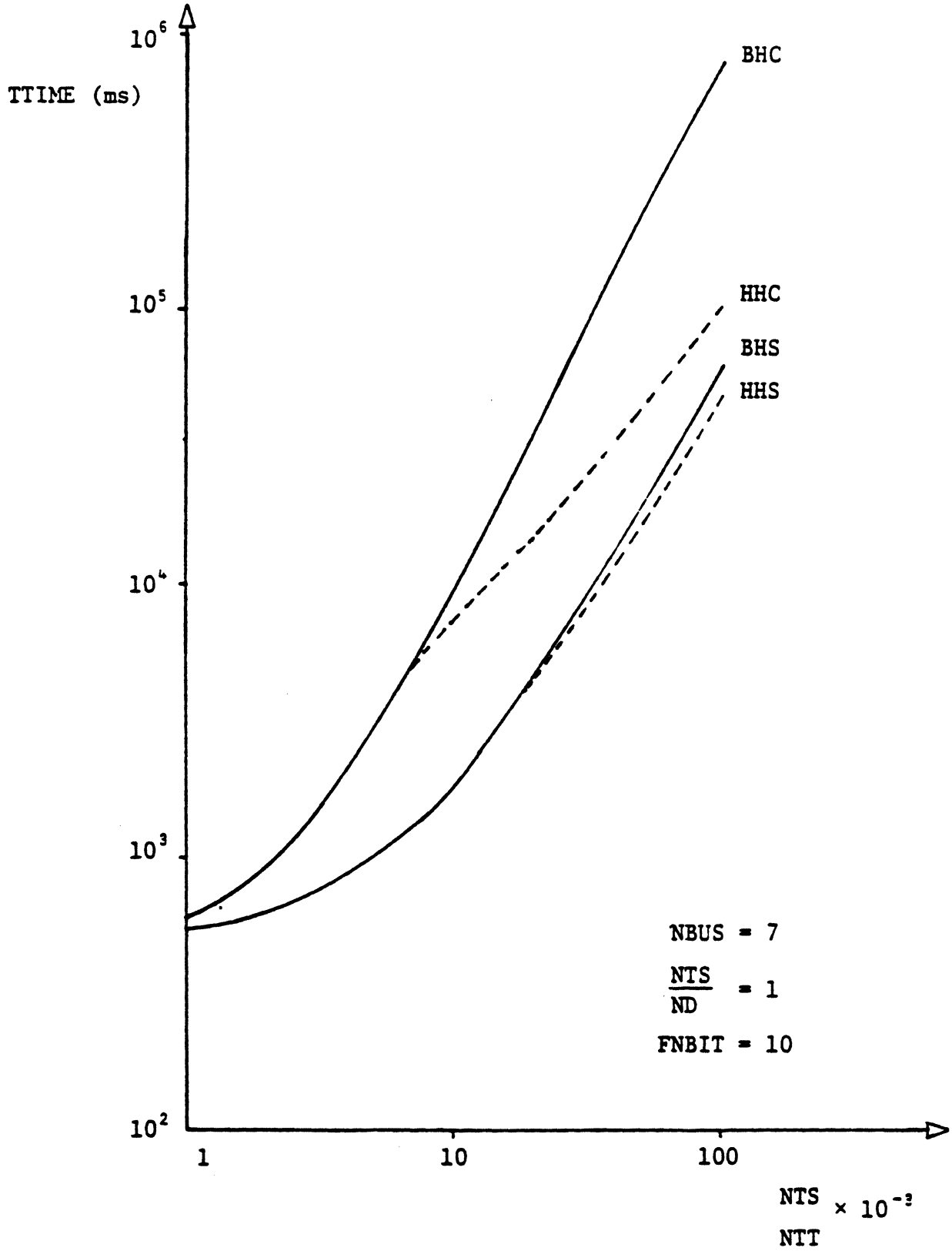

Figure 4.31 The Performance of the "Local Hash" STCF Algorithms

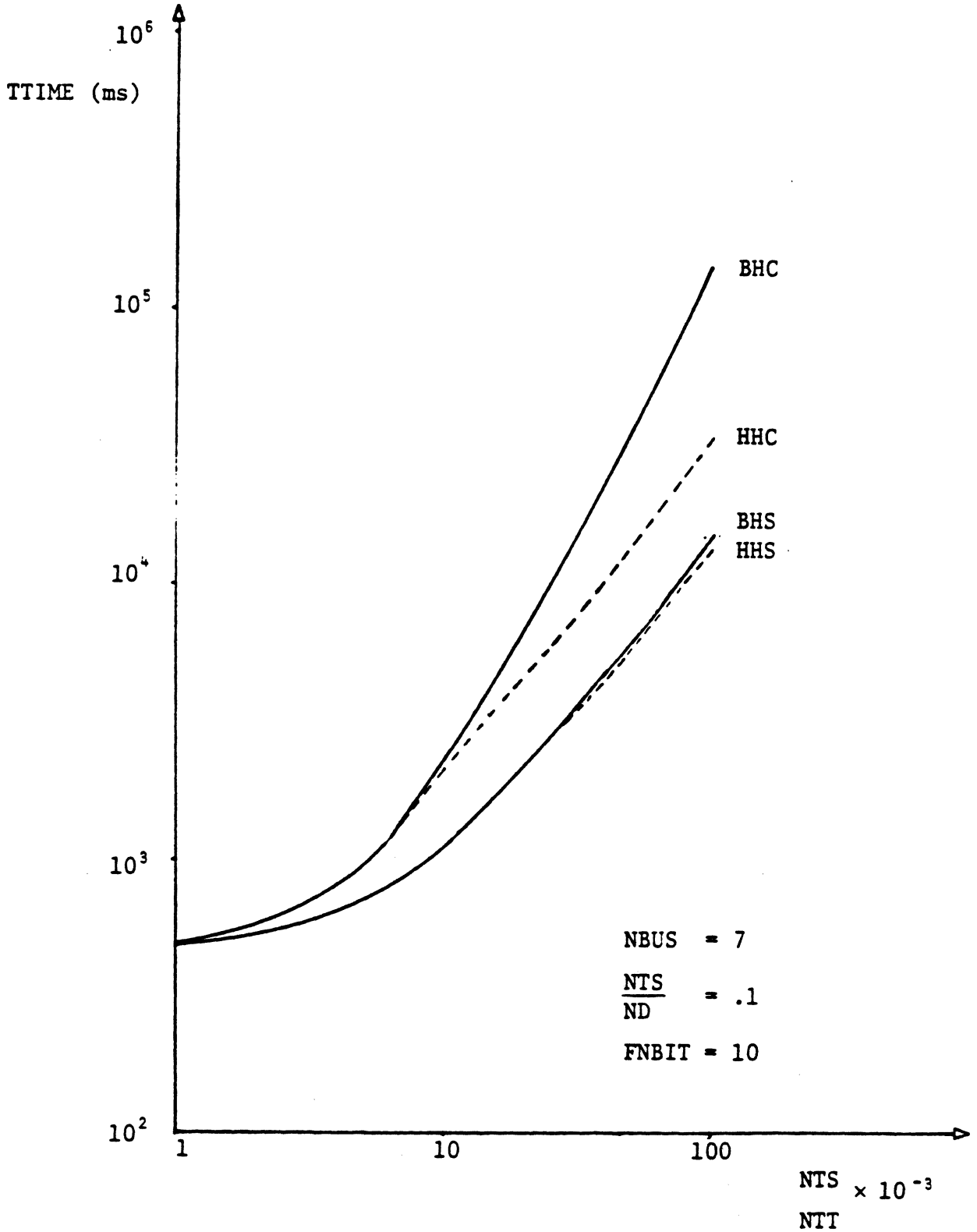

Figure 4.32 The Performance of the "Local Hash" STCF Algorithms

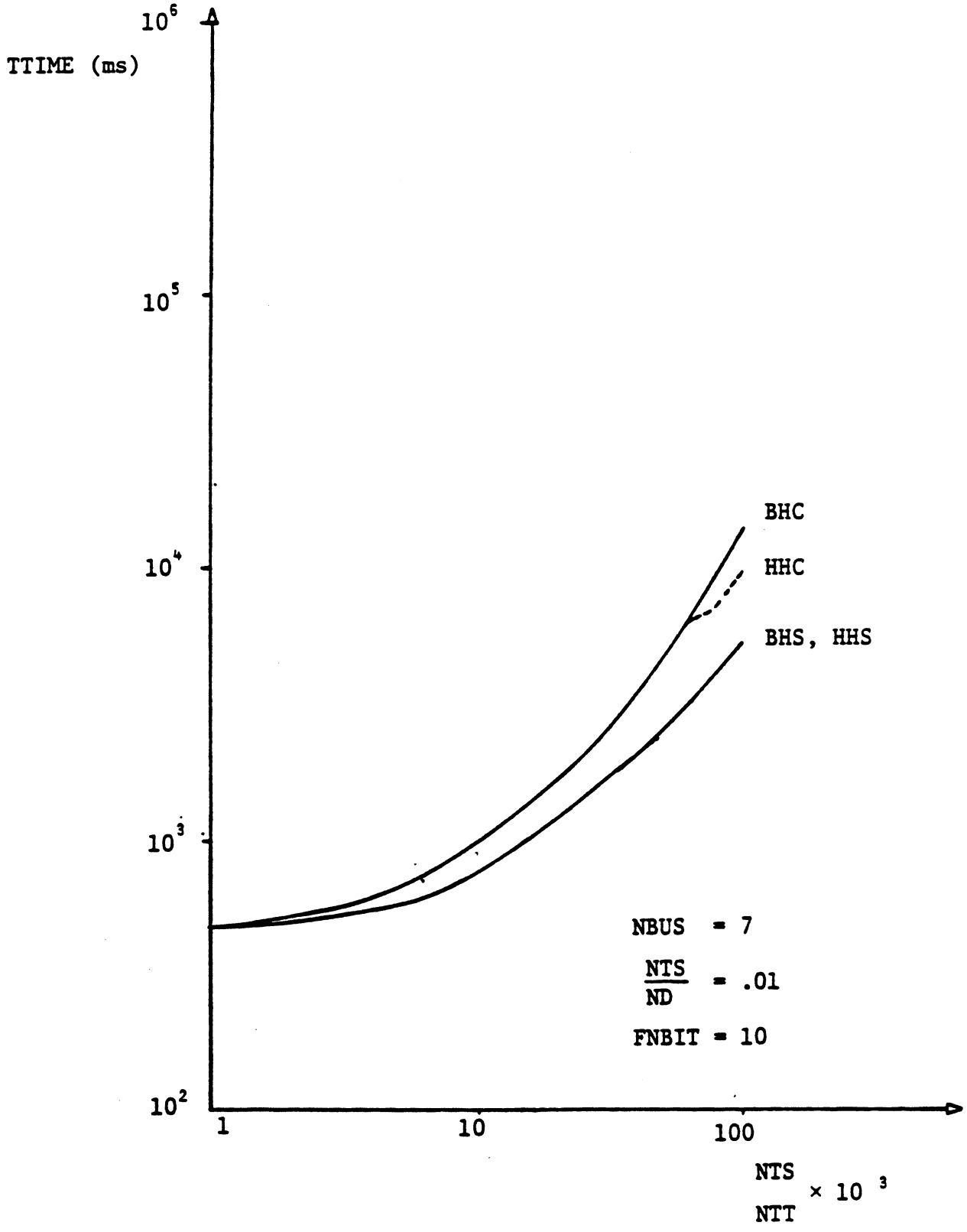

Figure 4.33 The Performance of the "Local Hash"

STCF Algorithms

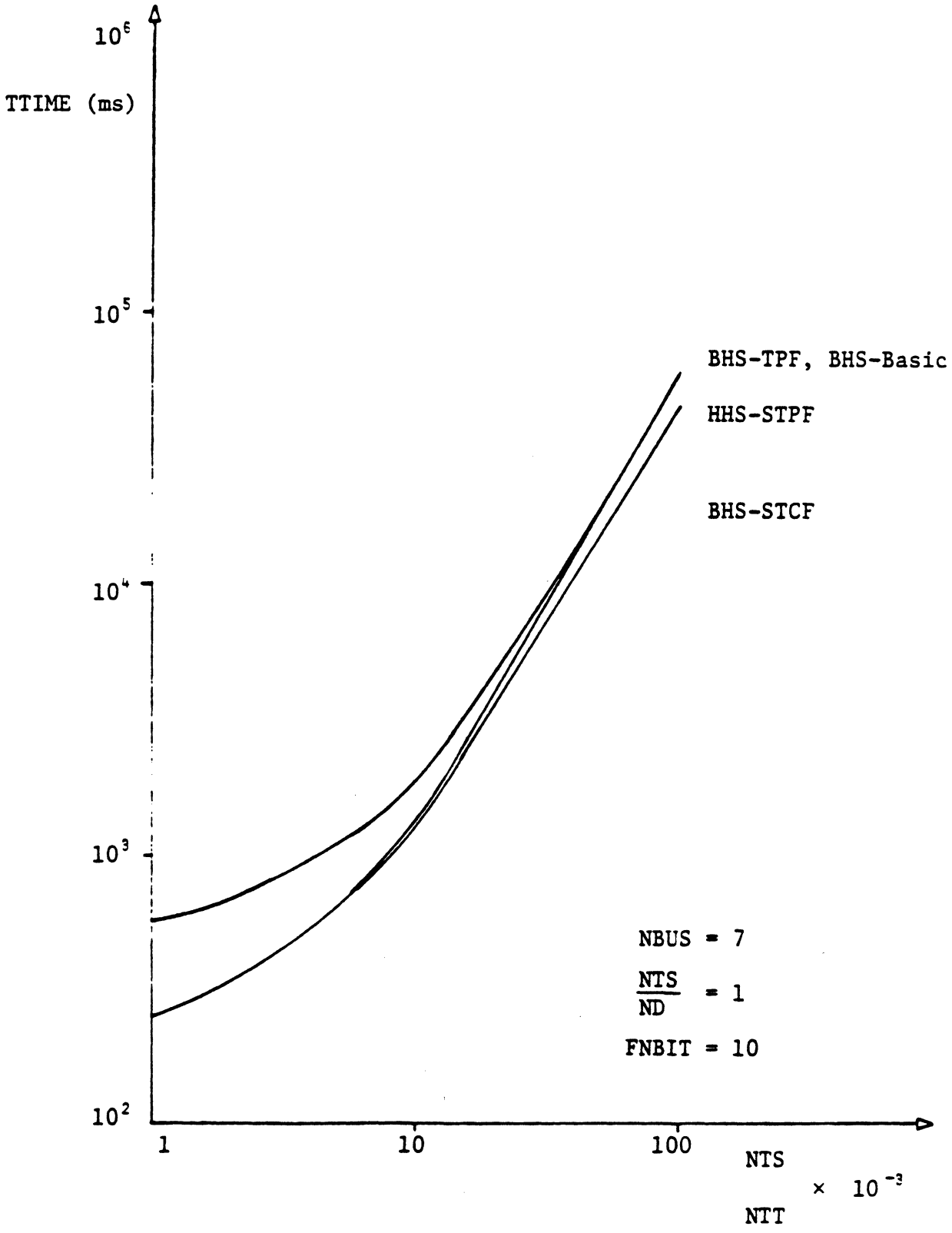

Figure 4.34 The Best Performing "Local Hash" Algorithm Within Each Algorithmic Category

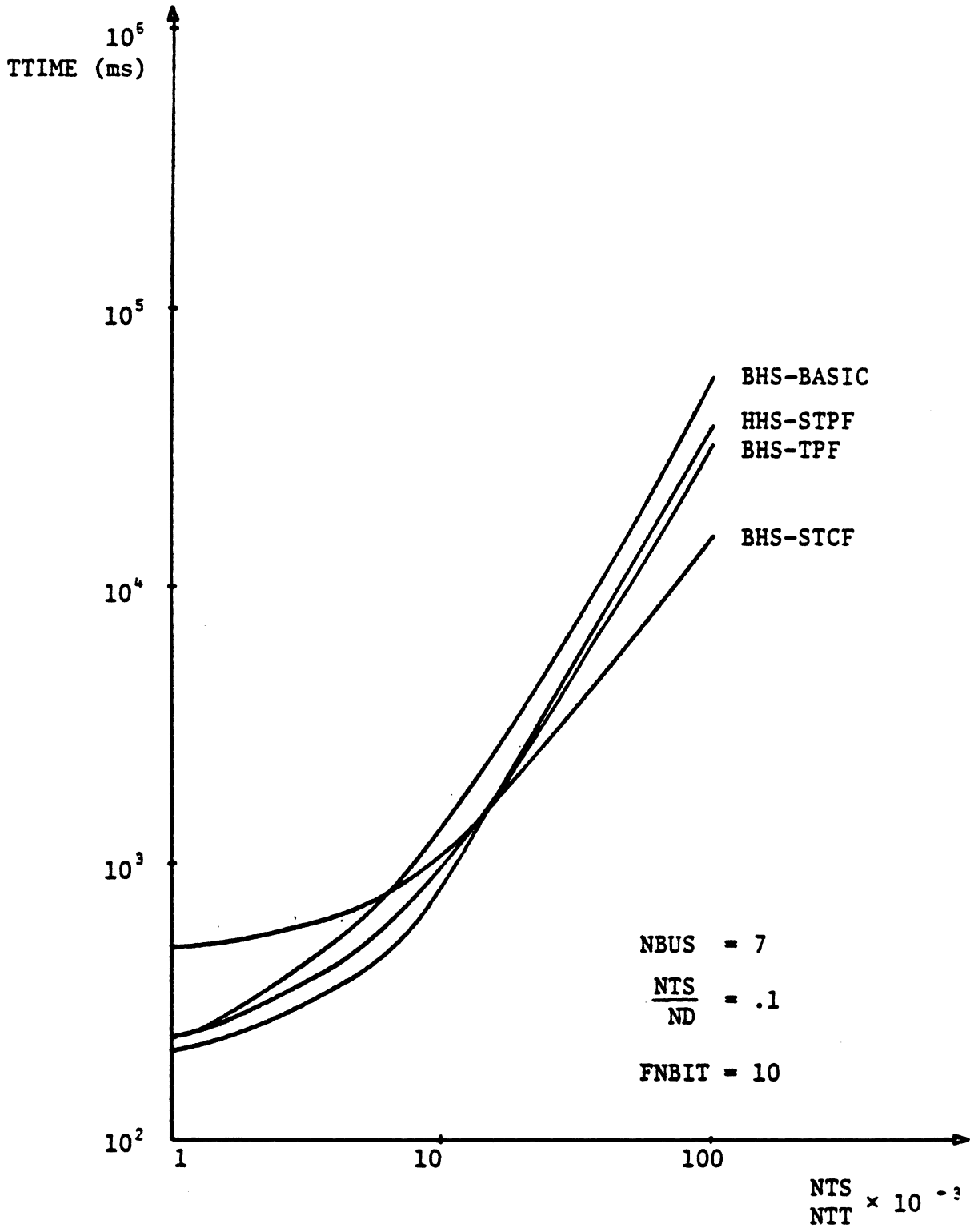

Figure 4.35 The Best Performing "Local Hash" Algorithms Within Each Equi-Join Algorithmic Category

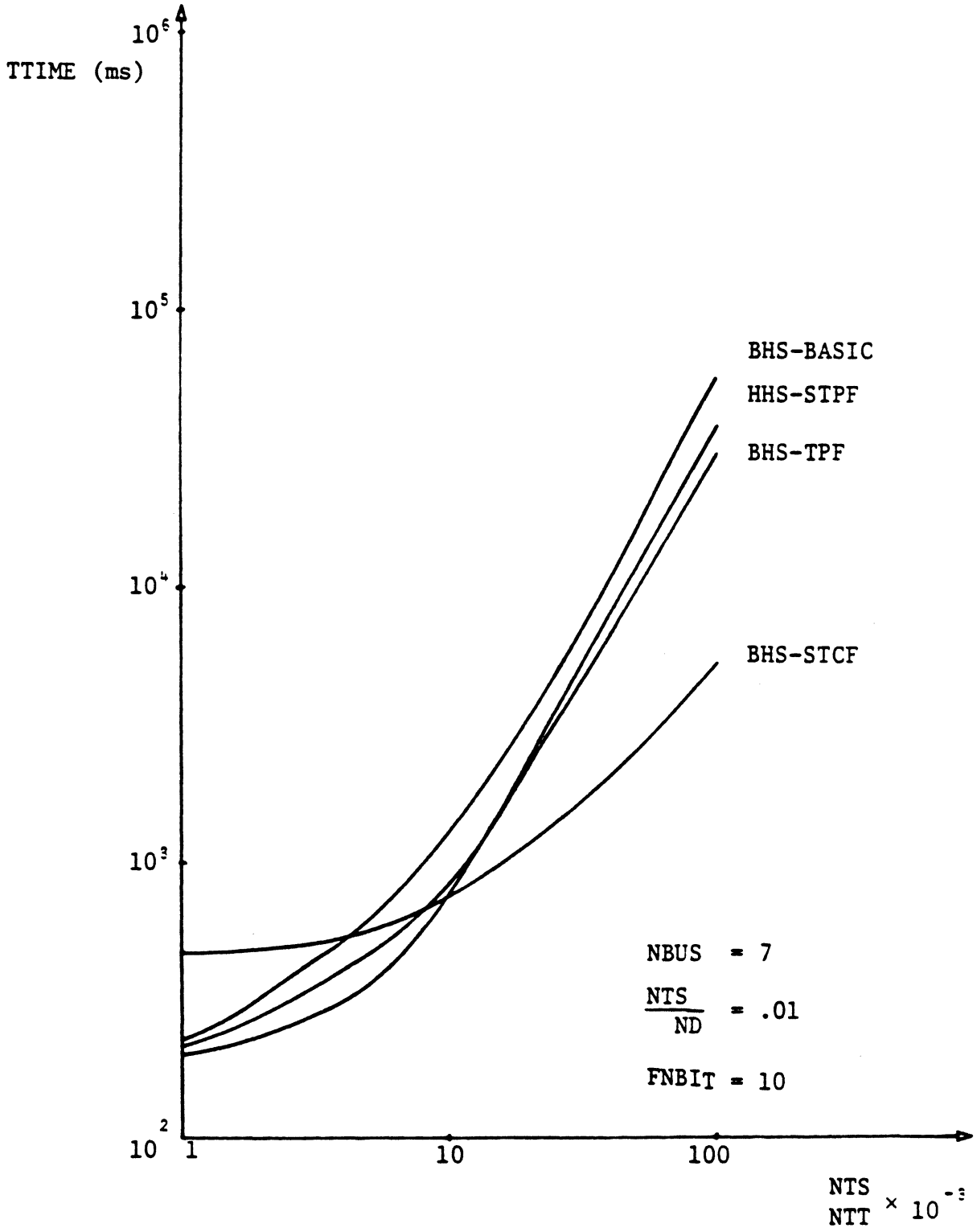

Figure 4.36 The Best Performing "Local Hash" Algorithm Within Each Algorithmic Category

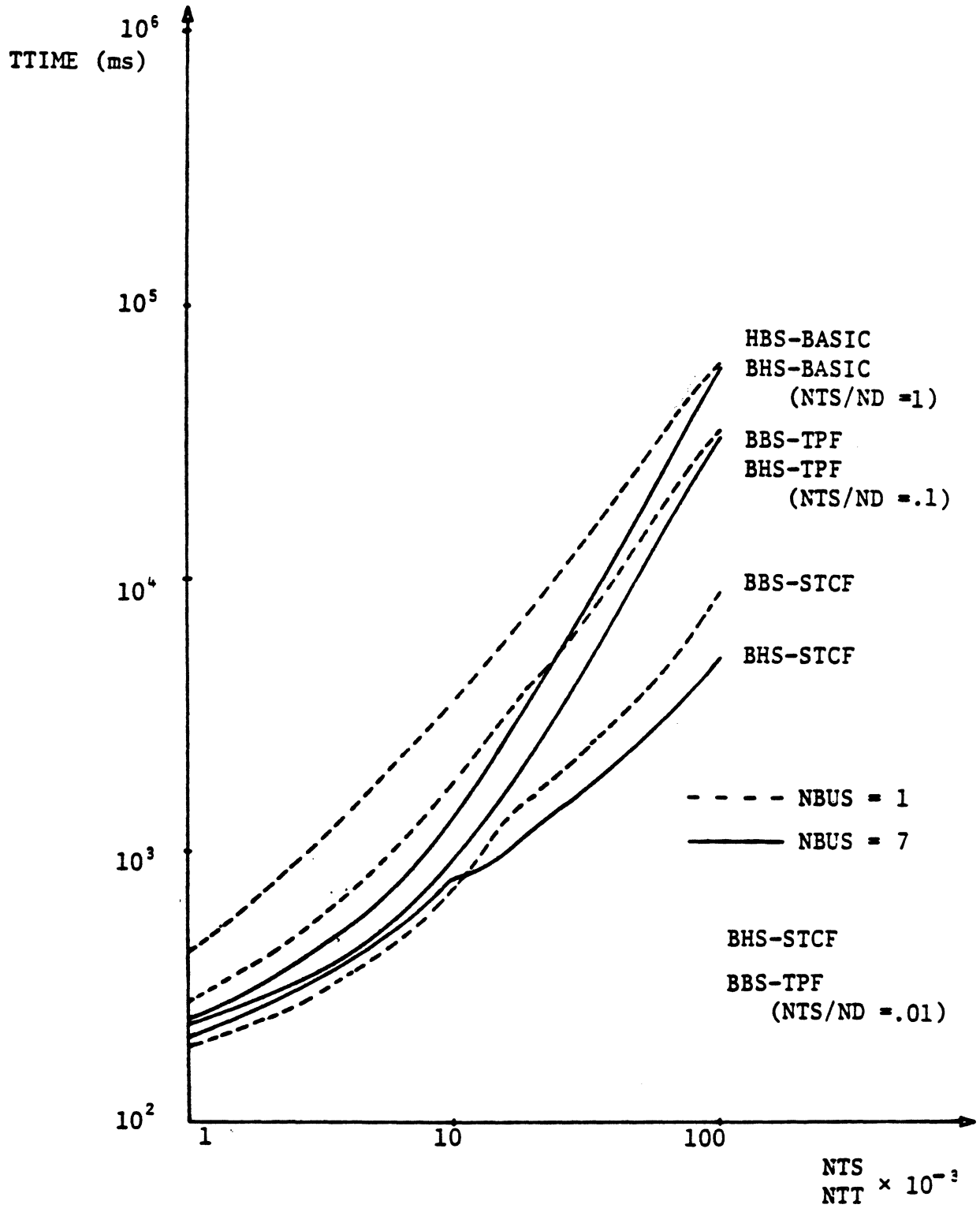

Figure 4.37 The Comparison Between the Best Performing Algorithms on the Two Architectures

respectively. From these figures, it can be concluded that the best performing algorithm is a function of the parameter [NTS/ND]. For [NTS/ND] = 1, the BHS-Basic is the best performing one (Figure 4.34). For [NTS/ND] = .1, the BHS-TPF is the best performing one (Figure 4.35). Finally, for [NTS/ND] = .01, the best performing algorithm is a composite one (Figure 4.36). For relations with small cardinalities, the BHS-TPF algorithm is the best performing one. On the other hand, the best performing one for relations with large cardinalities is the BHS-STCF algorithm.

In Figure 4.37 the performance parameter TTIME of the best performing algorithms is plotted for both the NBUS = 1 and NBUS=7 cases verses NTS(NTT) for NTS/ND ε {1, .1, .01}. From this Figure, the following important conclusions can be drawn:

1. For large values of the joining probability (NTS/ND ~ 1), or in the absence of the vector, the adoption of the multiple bus organization together with the BHS-Basic algorithm will improve the equi-join performance over that executed on the single bus organization. However, this improvement becomes insignificant when joining relations with large cardinalities.
2. For moderate to small values of the joining probabilities (NTS/ND < .1), the adoption of the multiple bus organization will either slightly improve the equi-join operation performance or not improve it at all.

From above and from the fact that most equi-join operations tend to have low joining probabilities, it can be concluded that the single bus organization is cost-effective in executing the equi-join operation, especially in the context of the very large database environments.

CHAPTER 5

ALGORITHMS FOR THE SELECTION, INDEX-SELECT AND PROJECTION

In general, selection and projection are fundamental operations to the relational databases. In most retrieval queries, the projection operation immediately follows the selection one. For performance improvement, the RDBM combines the latter two operations (whenever that is possible) to form a new operation, the selection-projection (SP). This operation is processed as a nondecomposable operation.

The newly proposed RDBM supports the execution of the selection, the projection and the selection-projection operations. It defines an index, the MAU index, on the most frequently referenced attributes in the database, together with an index retrieval operation, the index-select. In most cases, this structure will reduce the number of MAUs that the RDBM has to process for a given selection or selection-projection operation. In addition, the proposed RDBM uses parallel processing and algorithms for executing the latter as well as the index-select operations.

In executing the index-select operation, the RDBM uses one PC. On the other hand, it uses one or more PCs in executing the selection, the projection or the selection-projection operation. In general, the number of PCs allocated to execute one of the these operations is a master back-end controller (MBC) decision. This decision is based on many factors, such as the operation type, the size of the input relation, the expected size of the output relation, the number of available PCs and the priority class to which the operation's query belongs.

In this chapter, the set of algorithms which implement the selection, the projection and the index-select operations on the proposed RDBM are outlined.

5.1. The Selection-Projection Operation

The selection-projection operation is formally defined as follows:

$$SP_{QE}{}^{A}(R) = \left\{ r[A] : r \varepsilon R \ \cap \ QE(r) \right\}$$

where R is the input relation, A is a subset of relation R attributes and QE is a qualification expression (for its definition refer to Section 1.3).

SP processing proceeds in two phases. In the first phase, the set of MAU addresses, MA, containing the tuples which satisfy QE is found. In the second phase, the MAUs whose addresses are in the set MA are processed to obtain the operation resulting relation.

Recall that the proposed RDBM defines two index tables, the relation index and the MAU index. During the first phase, the MBC uses the information attached to the operation, during the compilation phase of its query (refer to Section 3.3.1), for the index to be used. If the relation index is to be used, then it will be accessed by the MBC for retrieving all the MAU addresses which store the tuples of the relation referenced by the SP operation. On the other hand, if the MAU index is to be used, then it will be accessed using the index-select operator. This operator will then retrieve (as is shown later) the set of MAU addresses which contain the tuples relevant to the SP operation.

During the second phase, the MBC determines the set of PCs for processing the set of MAUs of the first phase. The MBC distributes the operation code, the referenced relation format together with other necessary information, to the participating PCs. Having received the latter information, the CMP of each PC participating in executing the SP operation broadcasts such information to

every one of its triplets. The algorithm executed by the PCs for this phase is based on hashing. A hashing function, computed by the TPs of the participating PCs, partitions the tuples in the MAUs, based on the values of the attribute subset A, into disjoint subsets of tuples (referred to as the global subsets). The tuples of each of these subsets which satisfy QE must fit in a PC's LMUs. Then, in parallel, each PC processes a different subset. If the number of subsets is larger than that of the PCs, then the latter process is repeated until all the subsets have been processed.

The tuples, in a typical subset, are processed by further partitioning into disjoint subsets of tuples (referred to as the local subsets). The tuples of each of these subsets which satisfy QE must fit in the LMU of a triplet. Finally, every triplet projects its share of the selected tuples by sorting them and deleting the duplicate ones.

A typical participating PC collects and processes the tuples of a subset (say subset i) as follows:

1. The CMP requests, from the MBC, one MAU with an address in MA every time its triplets are through processing the current one. The MBC replies with the address of a PB block which stores an unprocessed* MAU (if the PB does not contain a relevant MAU, then the MBC requests that one be brought from the MM to PB). The CMP then directs its IOCs to read the corresponding PB block into its LMUs. This step is repeated for all MAUs whose addresses are in MA.

2. Every triplet (say Triplet n, for example) of the PC processes a typical tuple by first computing a hashing function over its value of the attribute subset A. The output of the function is a global subset index j. If $j \neq i$, then the tuple will be ignored, otherwise, the QE will be evaluated for the tuple.

*By "unprocessed" we mean "never processed by the PC requesting the MAU."

If QE evaluates to "false," then the tuple will be ignored. If the QE evaluates to "true," then only the value of the attribute subset A will be retained. Also, another hashing function will be computed over the attribute subset A. The output of the function is a local subset index (a triplet number) k.

3. If n = k, then the new tuple will be compared, by the triplet, to a duplicate free, sorted list of projected ones (constructed from previously processed tuples) using the binary search method[KNUT73]. If the list contains a duplicate tuple, then the new tuple will be ignored, otherwise, it will be inserted in the list in a sorted fashion.

4. If n ≠ k, then the data mover moves the new tuple (over the TBUS) to the k^{th} triplet. The latter triplet will then process the received tuple against its own duplicate free sorted list as in Step 3.

5.2. The Projection Operation

The execution of the projection operation on the proposed RDBM is carried out using a slightly modified version of the algorithm implemented for the SP operation. During the first phase in executing the projection algorithm, the set of MAU addresses which store all the tuples of the relation referenced by the operation is determined.

During the second phase, the same steps as those of the SP operation are carried out with a small difference, Step 2 of this phase does not include the evaluation of the QE (simply because there is no selection operation to be performed).

5.3. The Selection Operation

The proposed RDBM executes the Selection operation in two phases. The first phase is identical with that of the selection-projection operation. The

result of executing the latter phase is the set of MAU addresses MA relevant to the selection operation.

During the second phase, the MBC determines a set of PCs for executing the operation. The MBC distributes the operation code, the referenced relation format together with some other information, to the selected set of PCs. The MBC also requests some MAUs whose addresses are in the set MA to be brought into the parallel buffer. On the other hand, each CMP of a PC participating in the selection broadcasts the information, received from the MBC, to every one of its triplets. It also requests, from the MBC, one MAU each time its triplets are through processing the current one. The MBC replies with the parallel buffer block address which stores an unprocessed* MAU. The CMP directs its IOCs to read the corresponding parallel buffer block into their input buffers. Every triplet processor then searches its input buffer and retrieves those tuples which satisfy the selection qualification expression.

The tuples retrieved by the cluster's triplets are stored in their output buffers. Whenever the output buffers of a PC's triplets get full, the corresponding master processor requests an empty parallel buffer block. The MBC responds with a block address to which the cluster master processor directs its triplets' IOCs for storing their output buffers.

Although the algorithm, presented above for the selection operation, looks similar to that of DIRECT [DEWI79], nevertheless, the two algorithms differ from each other in several ways, namely:

1. In DIRECT, each query processor participating in the selection operation must go through the back-end controller (BEC) each time it inputs or outputs data. The overhead associated with such a scheme is shown by BORAL

*By "unprocessed" we mean "never processed by any of the PCs participating in executing the operation."

[BORA81] to create a system bottleneck at the BEC. In the algorithm presented above, whenever the triplets of a PC need to input or output data, the CMP, representing all these triplets, needs to request the MBC service only once. This reduces the algorithm overhead (by a factor equal to the number of triplets within a PC) and prevents the MBC from becoming a system bottleneck.

2. The algorithm for the selection operation presented above takes advantage of the MAU index to limit the number of MAUs which must be moved from the secondary storage and processed by a PC. For large relations and localized reference to the data this scheme largely improves the selection operation performance. On the other Hand, the DIRECT selection algorithm uses only a relation index. This results in poor selection performance especially in the large database environment.

5.4. The Index-Select Operation

The index-select operation is used in conjunction with a selection or selection-projection operation. It manipulates the index-term index to retrieve the set of MAU addresses which contain tuples relevant to the associated operation. The RDBM implements the Index-Select operation as follows:

1. The MBC selects one PC to execute the index-select operation. It also sends to the corresponding CMP of the chosen PC the operation code, the MQE and the index term format.

2. Receiving the above information, the CMP of the chosen PC broadcasts it to its triplets.

3. Every triplet of the PC initializes a hash table HTAB with a suitable number of entries. Each entry, in the latter table, contains a pointer to a link list. This list can contain zero or one or more nodes. Figure 5.1 shows the node format. The BITS field consists of several subfields, each corresponds to a predicate conjunction of MQE. A subfield consists of a set of bits, each of which corresponds to a predicate of the associated predicate conjunction. In order to improve the BITS field processing, the subfields must be aligned in memory at the byte boundaries.

4. The PC requests one IMAU of those referenced by the index-select operation, whenever it is through processing the current one. The MBC responds with the address of the parallel buffer block which stores an unprocessed MAU. The PC's triplets are then directed to read the parallel buffer block into their input buffers. This step is repeated until all the IMAUs relevant to the index-select operation have been processed.

5. Every triplet processor of a PC scans the tags of its share of the index-term blocks and compares the relation and attribute names associated with these tags to those of every predicate in MQE. Whenever a match occurs, the index terms of the corresponding block are processed one at a time as follows:

   The predicate is evaluated on the value part of the index term. If the result is true, then the MAU address field is augmented with the predicate number and hashed based on the MAU-Address, to one of the PC's triplets. The PC's master processor then moves it to the destination triplet. The processor of the destination triplet hashes the received MAU-Address to HTAB. If the MAU-Address is not already in the HTAB, then it is stored in a node and the proper bit in the BITS field of such node is set, otherwise, the proper bit in the node, where the MAU-Address is stored, is set.

6. When all the IMAUs relevant to the index-select operation have been processed, then the processor of every triplet in the PC does the following:

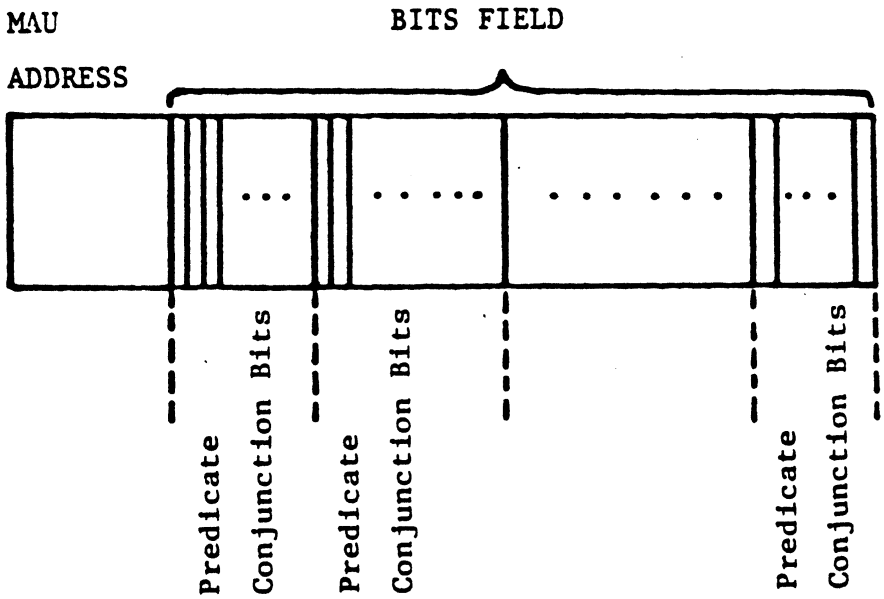

Figure 5.1 The Format of a Node in the Index-Select Processing Scheme

For each node in HTAB, evaluate the BITS field. If one of the node subfields evaluates to one then send the corresponding MAU-Address to the CMP.

7. The CMP of the chosen PC sends the list of addresses, accumulated in Step 6, to MBC.

One variation to the above algorithm would be to use the sorting and searching method instead of the hashing table method of Step 5. In the light of the high cost of computing a hashing function using today's microprocessors, the sorting method may have an average performance similar to that of the HTAB scheme, but a much better worst case one.

CHAPTER 6

CONCLUSIONS

6.1. Summary of Research

The general framework of the research presented in this thesis is the design of a back-end database machine (DBM) suitable for supporting concurrent, on-line, very large relational database systems. A structured approach was followed in designing such a machine. First, the relational data model, together with its most important operations and the previously proposed DBMs, were reviewed. Next, this review, coupled with the set of the very large database system requirements and the restrictions imposed by the current and anticipated state of the hardware technology, were used to formulate a set of design guidelines. Consequently, an architecture for a cost-effective database machine that meets this set of guidelines was synthesized.

In Chapter 1, the design process was began by reviewing the relational data model together with its most fundamental operations.

In Chapter 2, the design process was continued by reviewing the previously proposed DBMs. This review was guided by a novel classification scheme. The new scheme rests on three attributes, namely, the indexing level, the query Processing place and the processor-memory organization. The attribute "indexing level" describes the type of indexing a DBM supports for improving the processing of the most fundamental operation in the database system, namely, the selection (from the permanent relations of the database). Three levels of indexing are supported, namely, the database level (no indexing), the

relation level and the page level.

The attribute "query processing place" describes the place where a DBM executes the different parts of a user query. Some of the DBMs execute all the query on the disk (where the database is stored). Others execute all the query off the disk(in a separate processor-memory complex). A third group of DBMs execute part of the query On the disk and the other part off the disk (hybrid-DBMs).

The attribute "processor-memory organization" describes the way a DBM executes the database operations. Some DBMs have SISD organization, others have SIMD organization. A third group have MIMD organization.

Also, in Chapter 2, the novel scheme was used together with the current and anticipated state of technology to qualitatively analyze and compare the previously proposed DBMs. Some important conclusions regarding their cost-effectiveness were reached and presented.

In the last section of Chapter 2, the above conclusions coupled with the requirements of the large database systems are used to arrive at a set of guidelines along which the new machine was designed. These guidelines involve the use of the moving-head disk as a unit for mass storage, supporting the page level indexing and adopting both the off-disk and the MIMD organizations.

In chapter 3, a back-end DBM designed to meet the set of guidelines of Chapter 2 was presented. This machine is organized as a set of "processing clusters" managed and controlled by the master back-end controller (MBC). A processing cluster is organized as a set of SIMD processor-memory units (referred to as triplets) managed and controlled by the cluster master processor (CMP). Communications between the CMP and the triplets and between the triplets themselves are carried out over two buses, namely, the master bus (MBUS) and the triplet bus (TBUS). The database is stored on a set of

moving-head-disk units. Both, the set of processing clusters and the set of moving-head disks are interfaced to a set of memory blocks [referred to as the parallel buffer (PB) and can be implemented using the newly emerging electronic disk technology]. Each block has enough capacity to store the unit of data transfer (MACU) between the moving-head disks, the processing clusters and the parallel buffer. The interface allows two or more processing clusters or moving-head disks to read/write different parallel buffer blocks simultaneously. Also, it allows any set of processing clusters and moving-head disks to read the same parallel buffer block simultaneously. The interface is a modified version of the cross-point switch proposed by Dewitt[DEWI79]. In comparison with the latter one, our interface has a lower logic complexity.

The new machine also supports an indexing structure (page level indexing) to reduce the number of MAUs which must be processed for a typical selection operation. It is also provided with the ability to execute one or more database operations (from the same query or different ones) in parallel.

In Chapter 4, an extensive and detailed study was presented for implementing an important relational algebra operation, the equi-join operation, on the new DBM. A large set of algorithms was developed for such implementation. These algorithms were classified according to three important attributes. The first attribute specifies the way a given algorithm distributes the tuples, participating in the equi-join operation, among the processing clusters for processing. The second attribute specifies the way a given algorithm distributes the tuples of one processing cluster among its triplets for processing. The third attribute specifies the way a triplet performs the equi-join operation on its own tuples. The latter scheme was used in presenting the equi-join algorithms.

In Chapter 4, an analytical framework was also developed to evaluate and compare the different equi-join algorithms. This comparison was carried out

mainly through comparing the behavior of an important performance measure, namely, the "total execution time (TTIME)." The TTIME is the time to execute an equi-join algorithm on the new machine without any overlapping between the activities of its different units. To compute this performance measure for the various equi-join algorithms, a series of analytical, average valued models were introduced. Each one, called the execution model, models the execution of one of the latter algorithms on the proposed machine.

The above framework enabled us to determine the best performing equi-join algorithm. Which equi-join algorithm is the best performing one was found to depend on the characteristics of the data participating in the equi-join operation. The proposed framework also allowed us to investigate the cost-effectiveness of adding more buses to the processing cluster for improving the data communication between its triplets. It was found that in a normal database environment, this addition does not improve the performance of the equi-join operation, especially in the context of the very large database systems.

Finally, in Chapter 5, a set of powerful parallel algorithms were developed and presented for implementing, on the proposed machine, the other operations of the relational data model together with a primitive essential for the support of the page level indexing .

6.2. Contributions

The research reported in this thesis makes a number of contributions to the database machine area. The most important ones are:

1. The introduction of a novel scheme for the classification of the DBMs. it is believed that such a scheme is a step forward in the direction of finding a common framework for quantitatively comparing the different proposals

for a DBM. It also contributes to our understanding of the different DBM's organizations. It is also believed that such a scheme provides the researchers, in the DBMs area with an important tool to qualitatively analyze, compare and investigate the basic design trade-offs of the various proposals for the organization of the DBMs.

2. The introduction of a back-end DBM suitable for very large database system environment. The design of this machine rests on a set of principles. These principles include two fundamental ones followed by some previously designed DBMs, namely, the MIMD organization of DIRECT[DEWI79] and the "page level indexing" of DBC[BANE79]. While the MIMD organization is very valuable in handling the concurrent user environment, the "page level indexing" is equally important in supporting the very large database environment. The latter reduces the data volumes which need to be moved per selection/modification operation. As compared to the DBC, the newly proposed machine stores the database structure information on the relatively inexpensive mass storage devices (on moving-head disks rather than the more expensive electronic ones), manipulates the latter structure using the the same units (the processing clusters) which manipulate the database (thus distributing the systems workload uniformly among its various components) and provides the machine with the MIMD capability as well as the additional parallelism and processing power which are essential for meeting the requirements of the contemporary and anticipated very large database systems. Finally, the proposed architecture removes the restriction imposed by DBC on processing the equi-join operation [MENO81], namely, the fact that both the source and target relations of the equi-join operation must fit in the local memory of a processor-memory complex, designed specially to carry out the

corresponding operation. The newly proposed machine has the capability of joining relations of any sizes.

As compared to DIRECT, the proposed machine groups the processing elements into a set of clusters, each cluster with its own controlling processor. The data transfer in the new machine is done in relatively large units (the MACUs). This organization not only improves the management and control of the processing elements as well as reduces the overhead caused by processing the requests for the data units movement (experienced mainly by DIRECT), but also reduces the complexity of the interconnection network. Providing the new machine with "page level index" as well as supporting its primitive operation, at the hardware level, helps to improve the performance of the Selection operation. This operation performs poorly, on DIRECT, relative to other DBMs[HAWT81]. In the context of the very large databases, the new machine permits the implementation of a set of algorithms for the equi-join operation which is more powerful than the one implemented in DIRECT. In [QADA83A], the performance of the equi-join operation in both DIRECT and the new DBM using the modeling framework of chapter 4 are compared. It was found that in most database environments, the performance of the equi-join operation on the new machine is two to five times faster than that on DIRECT.

3. The introduction of an analytical framework for modeling the execution of the different equi-join algorithms on the new machine. It is believed that such modeling framework is general enough to be applicable for evaluating the execution of the equi-join algorithms on other proposed DBMs. This became clear when we saw the ease of using the new modeling technique in comparing the execution of the equi-join operation on DIRECT[QADA83A] to that on the new machine.

6.3. Further Research

There are several avenues of research to be explored based on this research, among these are:

1. Investigating the impact of the newly emerging VLSI technology on the design of the DBMs. It is believed that the scheme of Chapter 2 can aid in such an investigation.

2. Investigating the support of the update operations on the proposed machine. This will involve the design of two important mechanisms, namely, the tuples and index terms clustering mechanism and the concurrency control mechanism. While the first improves the storage as well as the processing efficiency, the second ensures the database integrity.

3 Developing simulation models for the execution of the best performing equi-join algorithms on the new machine. This will allow the DBM designer to closely investigate the trade-offs in the implementation the the latter algorithms.

4 Investigating the performance of the set of parallel algorithms proposed for implementing the selection, projection and index-select operations. It is believed that both analytical as well as simulation models can be found to carry out such evaluations.

APPENDICES

Appendix A

Derivations For the Equi-Join Execution Models

1.1. Some Derivations For the Execution Models of the Equi-Join Algorithms

In this appendix, some important derivations for the execution models of the equi-join algorithms are presented. These derivations rest on several important propositions. These propositions are stated and proofed below.

(1) Proposition 1

Given a collection of keys (possibly not distinct) with cardinality NK defined on a domain with cardinality ND. If a key is equally probable to have any value of the domain, then the expected number of distinguished values NDK in the collection of keys is:

$$NDK = ND(1-e^{-\frac{NK}{ND}}) \text{ for } NK, ND \gg 1$$

Proof:

This problem is similar to the classical occupancy problem treated by Feller [FELL67]. The occupancy problem deals with finding the number of empty cells m after throwing r balls to n cells. A ball is assumed to go to any cell with equal probability and the capacity of each cell is assumed to be infinite.

Feller showed that m has the following mass probability function.

$$P_m(r,n) = \binom{n}{m} \sum_{V=0}^{n-m} (-1)^V \binom{n-m}{V} \left(1-\frac{m+V}{n}\right)^r$$

He also showed that as $r,n \to \infty$, $P_m(r,n)$ can be computed from the following approximate formula:

$$P_m(r,n) = e^{-\lambda}\frac{\lambda^m}{m!}$$

where

$$\lambda \triangleq ne^{-r/n}$$

and the expected number of empty cells E_m is equal to λ.

Thus

$$\begin{aligned} E_{n-m} &= \text{expected number of occupied cells} \\ &= n - E_m = n - ne^{-r/n} \end{aligned}$$

$$E_{n-m} = n(1-e^{-r/n}) \tag{A.1}$$

Replacing n with ND and r with NK in Equation A.1 yields the following formula:

Expected number of distinguished values =

$$NDK = ND(1-e^{-\frac{NK}{ND}}) \tag{A.2}$$

Note

For finite NK and ND, the formula A.2 gives an approximate value for NDK. The error of this approximation approachs zero as NK and $ND \to \infty$. To see how well this approximation will perform for finite values of NK and ND, the expected number of distinguished values NDK using both the exact and the approximate formulas were calculated for two cases, namely, $ND = 10$, $NK = 10$ and $ND = 10$, $NK = 18$. The results are presented in Table A.1.

By analyzing the results of Table A.1, it is concluded that Equation A.2 is a good approximation for NDK even for relatively small values (>10) of ND and NK.

(2) Proposition 2

Table A.1 Exact and Approximate Values for NDK.

ND	NK	NDK Exact	NDK Approx.	% Error
10	10	6.5132	6.322	2.9
10	18	8.499	8.348	1.7

The expected number of keys per existing value is:

$$NK_v = \frac{NK}{NDK}$$

Proof:

Let Y be a random variable. Y is the number of keys which carry the same value:

$$NK_v = E[Y \mid Y \geq 1]$$

$$NK_v = \sum_{i=0}^{NK} ip(Y=i \mid Y \geq 1) \tag{A.3}$$

and

$$P(Y=i \mid Y \geq 1) = \frac{P(Y=i,\ Y \geq 1)}{P(Y \geq 1)}$$

$$\textit{since} \qquad P(Y \geq 1) = \frac{NDK}{ND} \text{ and } P(Y=0,\ Y \geq 1) = 0$$

$$\text{then} \qquad P(Y=i \mid Y \geq 1) = \frac{ND}{NDK} P(Y=i) \tag{A.4}$$

By combining Equations A.3 and A.4, the following formula for NK_v will result:

$$\begin{aligned} NK_v &= \frac{ND}{NDK} \sum_{i=0}^{NK} ip(Y=i) \\ &= \frac{ND}{NDK} E[Y] \end{aligned} \tag{A.5}$$

It is easy to show that Y has a binomial mass distribution function. That is,

$$P(Y=i) = \binom{NK}{i}\left[\frac{1}{ND}\right]^{i}\left[1-\frac{1}{ND}\right]^{NK-i}$$

and the $E[Y]$ is $\frac{NK}{ND}$. By Substituting $\frac{NK}{ND}$ for the latter quantity, in Equation A.5, the following formula will result.

$$NK_v = \frac{ND}{NDK}\cdot\frac{NK}{ND} = \frac{NK}{NDK}$$

(3) Proposition 3

Given a collection of keys (distinct) with cardinality NK. If a key hashes to the components of a cluster vector, of size NBIT, with equal probability, then the expected number of bits set (NBS) in the vector will be

$$NBS = NBIT\left[1-e^{-\frac{NK}{NBIT}}\right] \qquad \text{for } NK,\ NBIT \gg 1$$

Proof:

The proof follows that of Proposition 1. By replacing ND and NDK of Equation A.2 with NBIT and NBS, respectively, the above formula will result.

(4) Proposition 4

Given a collection of keys (distinct) with cardinality NK. If a key hashes to a hash table (with number of buckets NBP) with equal probability, then the expected number of buckets with one or more keys (NBF) is:

$$NBF = NBP\left[1-e^{-\frac{NK}{NBP}}\right] \qquad \text{for } NK,\ NBP \gg 1 \qquad \text{(A.6)}$$

and the expected number of keys per non-empty bucket NDK_B is:

$$NDK_b = \frac{NK}{NBF} \qquad \text{(A.7)}$$

Proof:

The proof of formula A.6 follows that of proposition 1. By replacing ND and NDK of Equation A.2 with NBP and NBF, respectively, formula A.6 will result.

The proof of formula A.7 follows that of proposition 2.

In Section A.1 the derivation of formulas to compute some important quantities, common to all the equi-join execution models, is presented. In Sections A.2 through A.5 the derivation of formulas to compute some important quantities are presented, respectively, for the basic, the TPF, the STPF and the STCF equi-join execution models.

1.2. Derivations Common to the Equi-Join Execution Models

In this section formulas for three important quantities are derived, namely, NMS, NMT and NMO.

Let "$\lceil\ \rceil$" denote the ceiling function and $x \in \{S,T\}$ then NMx, the number of MAUs which store the relation x, can be expressed as follows:

$$NMx = \left\lceil \frac{NTx \cdot LTx}{MAUC} \right\rceil \qquad \text{(A.8)}$$

The derivation of a formula for NMO proceeds as follows:

Let

NDS =

The expected number of distinguished values in the join attribute of the source relation.

NTS_v =

The expected number of source tuples with the same value.

The expected tuples in the output relation NTO is $[NTT \cdot NTS_v \cdot P$ (a tuple of the target relation carries, in its join attribute, a value which exists in the join attribute of the source tuples)].

Using Proposition 2, NTS_v can be expressed as $\frac{NTS}{NDS}$. Therefore,

$$NTO = NTT \cdot \frac{NTS}{NDS} \cdot \frac{NDS}{ND} = \frac{NTT \cdot NTS}{ND} \tag{A.9}$$

and

$$NMO = \left\lceil \frac{NTT \cdot NTS(LTS + LTT)}{ND \cdot MAUC} \right\rceil \tag{A.10}$$

1.3. Derivations for the Basic Equi-Join Execution Models

In this section and the following ones, the following notation is adopted:

Notation

NTS_M =

.the number of tuples in one MAU of the Source relation.

NDS_M =

The expected number of distinguished values in the join attribute of the tuples in one source MAU.

NTS_{MV} =

The expected number of tuples in one source MAU which has the same join attribute value.

NTS_G =

The expected number of source tuples which hash to one global bucket.

NDS_G =

The expected number of distinguished values in the join attribute of the source tuples of one global bucket.

NTS_{GV} =

The expected number of source tuples in one global bucket which have the same join attribute value.

NTS_P =

The expected number of source tuples in one triplet.

NDS_P =

The expected number of distinguished values in the join attribute of the source tuples in one triplet.

NTS_{PV} =

The expected number of source tuples in one triplet which have the same join attribute value.

NDS_B =

The expected number of distinguished values in the join attribute of the source tuples of one hash table bucket.

NTS_S =

The number of tuples stored in the same track of each source MAU.

NDS_S =

The number of distinguished values in the join attribute of the tuples stored in the same track of each source MAU.

In the following, the derivation of ENC(1) expressions for the BBH-, BHH-, HBH-, and HHH- basic equi-join algorithms are presented.

Notation

Let x ε {*BBH−Basic*, *BHH−Basic*, *HBH−Basic*, *HHH−Basic*}

ENCT(x)

is the expected number of comparisons a triplet processor will perform in joining one target tuple with its share of the source tuples during the execution of one phase of the x equi-join algorithm.

(a) Deriviation of ENC(1) for the BBH-Basic Equi-Join Algorithm

Let A be the event that a target tuple hashes to a non-empty bucket of the hash table.

Then,

$$ENCT(BBH-Basic) = \phi \cdot P(\bar{A}) + NTS_{PV} \cdot NDS_B P(A) \qquad (A.11)$$

Let E be the event that the target tuple's join attribute value has a match among those of the source tuples stored in the hash table.

Then,

$$P(A) = P(A \cap (E \cup \bar{E})) = P(A \cap E) + P(A \cap \bar{E})$$
$$= P(A \mid E)P(E) + P(A \mid \bar{E})P(\bar{E})$$

$$P(A) = 1 \cdot \frac{NDSp}{ND} + \frac{NBF^*}{NBP}\left[1 - \frac{NDSp}{ND}\right] \qquad (A.12)$$

The parameter ENC(1) of the BBH-Basic equi-join algorithm can be expressed as follows:

$$ENC(1) = NTT \cdot ENCT(BBH-Basic) \qquad (A.13)$$

By combining Equations A.11 through A.13, the following formula will result:

*In the derivation of P(A|E) the following is assumed: given that the target tuple's join attribute has no match among those of the triplet processor source tuples, the target tuple will hash to any of the hash table buckets with equal probability.

$$ENC(1) = NTT \cdot NTS_{PV} \cdot NDS_B \left\{ \frac{NDSp}{ND} + \frac{NBF}{NBP} \left[1 - \frac{NDSp}{ND} \right] \right\} \quad \text{(A.14)}$$

Using Propositions 1, 2 and 4, one can show that the following relations hold for the BBH-Basic equi-join algorithm:

$$NTSp = \frac{NTS}{NMS \cdot NP} \quad \text{(A.15)}$$

$$NTS_{PV} = \frac{NTSp}{NDSp} \quad \text{(A.16)}$$

$$NDS_B = \frac{NDSp}{NBF} \quad \text{(A.17)}$$

$$NDSp = ND\left[1 - e^{-\frac{NTSp}{ND}}\right] \quad \text{(A.18)}$$

and

$$NBF = NBP\left[1 - e^{-\frac{NTSp}{ND}}\right] \quad \text{(A.19)}$$

By combining Equation A.14 through A.19, the following expression for ENC(1) will result:

$$ENC(1) = \frac{NTT \cdot NTS}{NMS \cdot NP \cdot NBP} \left\{ \frac{1 - e^{-\frac{NTSp}{ND} \cdot e^{-\frac{NDSp}{NBP}}}}{1 - e^{-\frac{NDSp}{NBP}}} \right\} \quad \text{(A.20)}$$

(b) Deriviation of ENC(1) for the BHH-Basic Equi-Join Algorithm

Let

A be as defined for the BBH-Basic algorithm and

D be the event that a target tuple will hash to a triplet processor

The quantity ENCT(BHH-Basic) can be expressed as follows:

$$ENCT(BHH) = NTS_{MV} \cdot NDS_B P(A \cap D) \quad \text{(A.21)}$$

Let

E be the event that a target tuple's join attribute value has a match among those of the source tuples of one MAU.

Then,

$$P(A \cap D) = P(A \cap D \cap E) + P(A \cap D \cap \bar{E})$$

$$P(A \cap D \cap E) = P(A \mid D \cap E)\, P(D \mid E)\, P(E)$$

$$P(A \cap D \cap E) = 1 \cdot \frac{NDSp}{NDS_M} \frac{NDS_M}{ND} = \frac{NDSp}{ND}$$

and

$$P(A \cap D \cap \bar{E}) = P(A \mid D \cap \bar{E})\, P(D \mid \bar{E})\, P(E)$$

$$= \frac{NBF}{NBP} \cdot \frac{1}{NP} \cdot \left[1 - \frac{NDS_M}{ND}\right]$$

Therefore,

$$P(A \cap D) = \frac{1}{NP} \left\{ NP \cdot \frac{NDSp}{ND} + \frac{NBF}{NBP} \left[1 - \frac{NDS_M}{ND}\right] \right\} \qquad \text{(A.22)}$$

It is easy to show that for the BHH-Basic algorithm, the following equation is true:

$$NDSp = \frac{NDS_M}{NP} \qquad \text{(A.23)}$$

By combining Equations A.22 and A.23, the following formula for $P(A \cap D)$ will result:

$$P(A \cap D) = \frac{1}{NP} \left\{ \frac{NDS_M}{ND} + \frac{NBF}{NBP} \left[1 - \frac{NDS_M}{ND}\right] \right\} \qquad \text{(A.24)}$$

It is also easy to see that

$$ENC(1) = NTT \cdot ENCT(BHH-Basic) \tag{A.25}$$

By combining Equations A.21, A.24 and A.25, the following formula will result:

$$ENC(1) = \frac{NTT \cdot NTS_{MV} \cdot NDS_B}{NP} \left\{ \frac{NDS_M}{ND} + \frac{NBF}{NBP} \left[1 - \frac{NDS_M}{ND} \right] \right\} \tag{A.26}$$

Using Propositions 1, 2 and 4 shows that the following relations hold for the BHH-Basic equi-join algorithm:

$$NTS_M = \frac{NTS}{NMS} \tag{A.27}$$

$$NTS_{MV} = \frac{NTS_M}{NDS_M} \tag{A.28}$$

$$NDS_B = \frac{NDS_p}{NBF} = \frac{NDS_M}{NP \cdot NBF} \tag{A.29}$$

$$NDS_M = ND \left[1 - e^{-\frac{NTS_M}{ND}} \right] \tag{A.30}$$

and

$$NBF = NBP \left[1 - e^{-\frac{NDS_p}{NBP}} \right] \tag{A.31}$$

By combining Equations A.26 through A.31, the following expression for ENC(1) will result:

$$ENC(1) = \frac{NTT \cdot NTS}{NMS \cdot NP^2 \cdot NBP} \left\{ \frac{1 - e^{-\frac{NTS_M}{ND}} e^{-\frac{NDSp}{NBP}}}{1 - e^{-\frac{NDSp}{NBP}}} \right\} \tag{A.32}$$

(c) Deriviation of ENC(1) for the HBH-Basic Equi-Join Algorithm

Let

A be as defined before,

C be the event that a target tuple will hash to the current global bucket and

E be the event that a target tuple's join attribute value has a match among those of the tuples stored in the same track of each source MAU.

The quantity ENCT(BHH-Basic) can be expressed as follows:

$$ENCT(HBH-Basic) = NTS_{PV} \cdot NDS_B \cdot P(A \cap C) \tag{A.33}$$

$$P(A \cap C) = P(A \cap C \cap E) + P(A \cap C \cap \bar{E})$$

$$P(A \cap C \cap E) = P(A \mid C \cap E)P(C \mid E)P(E)$$
$$= 1 \cdot \frac{NDS_p}{NDS_S} \cdot \frac{NDS_S}{ND} = \frac{NDS_p}{ND}$$

and

$$P(A \cap C \cap \bar{E}) = P(A \mid C \cap \bar{E})P(C \mid \bar{E})P(\bar{E})$$
$$= \frac{NBF}{NBP} \cdot \frac{1}{NGB}\left[1 - \frac{NDS_S}{ND}\right]$$

Therefore,

$$P(A \cap C) = \frac{1}{NGB}\left\{NGB \cdot \frac{NDS_p}{ND} + \frac{NBF}{NBP} \cdot \left[1 - \frac{NDS_s}{ND}\right]\right\} \tag{A.34}$$

It is easy to see that for the HBH-Basic algorithm, the following equation is true:

$$NDS_p = \frac{NDS_S}{NGB} \tag{A.35}$$

By combining Equations A.34 and A.35, the following formula for $P(A \cap C)$ will result:

$$P(A \cap C) = \frac{1}{NGB}\left\{\frac{NDS_S}{ND} + \frac{NBF}{NBP}\left[1-\frac{NDS_S}{ND}\right]\right\} \quad \textbf{(A.36)}$$

It is easy also to see that

$$ENC(1) = NTT \cdot ENCT(HBH-Basic) \quad (A.37)$$

By combining Equations A.33, A.36 and A.37, the following equation will result:

$$ENC(1) = NTT \cdot NTS_{PV} \cdot NDS_B \cdot \frac{1}{NGB}\left\{\frac{NDS_S}{ND} + \frac{NBF}{NBP}\left[1-\frac{NDS_S}{ND}\right]\right\} \quad (A.38)$$

Propositions 1, 2 and 4 can be used to show that the following relations hold for the HBH-Basic equi-join algorithm:

$$NTS_S = \frac{NTS}{NP} \quad (A.39)$$

$$NTS_P = \frac{NTS_S}{NGB} \quad (A.40)$$

$$NTS_{PV} = \frac{NTSp}{NDSp} \quad (A.41)$$

$$NDS_B = \frac{NDSp}{NBF} \quad (A.42)$$

$$NDS_S = ND\left[1-e^{-\frac{NTS_S}{ND}}\right] \quad (A.43)$$

and

$$NBF = NBP\left[1-e^{-\frac{NDS_P}{NBP}}\right] \quad (A.44)$$

By combining Equations A.38 through A.44 together with Equation A.35, the following expression for ENC(1) will result:

$$ENC(1) = \frac{NTT \cdot NTS}{NP \cdot NBP \cdot NGB^2} \left\{ \frac{1 - e^{-\frac{NTS_S}{ND}} e^{-\frac{NDS_P}{NBP}}}{1 - e^{-\frac{NDSp}{NBP}}} \right\} \qquad \text{(A.45)}$$

(d) Deriviation of ENC(1) for the HHH-Basic Equi-Join Algorithm

Let

A, D and C be as defined before and

E be the event that a target tuple's join attribute value has a match among those of the source tuples:

$$ENCT(HHH-Basic) = NTS_{CV} \cdot NDS_B \cdot P(A \cap C \cap D) \qquad \text{(A.46)}$$

$$P(A \cap D \cap C) = P(A \cap C \cap D \cap E) + P(A \cap C \cap D \cap \bar{E})$$
$$P(A \cap C \cap D \cap E) = P(A \mid C \cap D \cap E)\, P(D \mid C \cap E)\, P(C \mid E)\, P(E)$$
$$= 1 \cdot \frac{NDSp}{NDS_C} \cdot \frac{NDS}{ND} = \frac{NDSp}{ND}$$

and

$$P(A \cap C \cap D \cap \bar{E}) = P(A \mid C \cap D \cap \bar{E})\, P(D \mid C \cap \bar{E})\, P(C \mid \bar{E})\, P(\bar{E})$$
$$= \frac{NBF}{NBP} \cdot \frac{1}{NP} \cdot \frac{1}{NGB} \cdot \left[1 - \frac{NDS}{ND}\right]$$

Therefore,

$$P(A \cap D \cap C) = \frac{1}{NP \cdot NGB} \left\{ NP \cdot NGB \cdot \frac{NDSp}{ND} + \left[1 - \frac{NDS}{ND}\right] \right\} \qquad \text{(A.47)}$$

It is easy to see that for the HHH-Basic algorithm the following equation is true:

$$NDSp = \frac{NDS}{NGB \cdot NP} \qquad \text{(A.48)}$$

By combining Equations A.47 and A.48, the following equation will result:

$$P(A \cap D \cap C) = \frac{1}{NP \cdot NGB}\left\{\frac{NDS}{ND} + \frac{NBF}{NBP}\left[1 - \frac{NDS}{ND}\right]\right\} \quad \textbf{(A.49)}$$

It is easy to see that

$$ENC(1) = NTT \cdot ENCT(HHH-Basic) \quad \text{(A.50)}$$

By combining Equations A.46, A.49 and A.50, the following equation will result:

$$ENC(1) = \frac{NTT \cdot NTS_{GV} \cdot NDS_B}{NP \cdot NGB}\left\{\frac{NDS}{ND} + \frac{NBF}{NBP}\left[1 - \frac{NDS}{ND}\right]\right\} \quad \text{(A.51)}$$

proposition 1, 2 and 4 can be used to show that the following equations are true for the HHH-Basic algorithm:

$$NTS_G = \frac{NTS}{NGB} \quad \text{(A.52)}$$

$$NDS_G = \frac{NDS}{NGB} \quad \text{(A.53)}$$

$$NTS_{GV} = \frac{NTS_G}{NDS_G} \quad \text{(A.54)}$$

$$NDS_P = \frac{NDS_G}{NP} \quad \text{(A.55)}$$

$$NDS_B = \frac{NDSp}{NBF} \quad \text{(A.56)}$$

$$NDS = ND\left[1 - e^{-\frac{NTS}{ND}}\right] \quad \text{(A.57)}$$

and

$$NBF = NBP\left[1 - e^{-\frac{NDSp}{NBP}}\right] \quad \text{(A.58)}$$

By combining Equations A.51 through A.58, the following expression for ENC(1) will result:

$$ENC(1) = \frac{NTT \cdot NTS}{NGB^2 \cdot NP^2 \cdot NBP}\left\{\frac{1-e^{-\frac{NTS}{ND}}e^{-\frac{NDSp}{NBP}}}{1-e^{-\frac{NDSp}{NBP}}}\right\} \qquad (A.59)$$

1.4. Derivations for the TPF Equi-Join Execution Models

In this section the derivations of the formulas for the parameters LTB and LTG are presented. These parameters are used in developing the "global broadcast" TPF and the "global hash" TPF execution models, respectively. Also, the derivation of the ENC(1) expressions for BBH-TPF, BHH-TPF, HBH-TPF and HHH-TPF equi-join algorithms are presented.

Notation

Let $x \in \{BBH-TPF, BHH-TPF, HBH-TPF, HHH-TPF\}$

ENCT(x) be as defined before

(a) Deriviation of the LTB Formula

Let

B be the event that a target tuple will hash to a "set" bit in the cluster vector and

E be the event that a target tuple's join attribute value has a match among those of the source tuples in one MAU:

$$LTB = P(B) \qquad (A.60)$$

$$P(B) = P(B \cap E) + P(B \cap \bar{E})$$

$$P(B) = P(B \mid E)P(E) + P(B \mid \bar{E})P(\bar{E})$$

and

$$P(B) = 1.\frac{NDS_M}{ND} + \frac{NBS}{NBIT}\left[1-\frac{NDS_M}{ND}\right] \qquad (A.61)$$

Using Proposition 3, it be can shown that the following equation is true for the "global broadcast" TPF equi-join algorithms:

$$NBS = NBIT\left[1-e^{-\frac{NDS_M}{NBIT}}\right] \tag{A.62}$$

By combining equations A.60 through A.62, the following expression for LTB will result:

$$LTB = 1-e^{-\frac{NTS_M}{ND}}e^{-\frac{NDS_M}{NBIT}} \tag{A.63}$$

where NTS_M and NDS_M are computed using equations A.27 and A.30, respectively.

(b) Deriviation of the LTG Formula

Let

B,C be defined as before and

E be the event that a target tuple's join attribute value has a match among those of the source tuples

$$LTG = P(B \cap C) \tag{A.64}$$

$$P(B \cap C) = P(B \cap C \cap E) + P(B \cap C \cap \bar{E})$$
$$P(B \cap C \cap E) = P(B \mid C \cap E)P(C \mid E)P(E)$$
$$= 1 \cdot \frac{NDS_G}{NDS} \cdot \frac{NDS}{ND} = \frac{NDS_G}{ND}$$

and

$$P(B \cap C \cap \bar{E}) = P(B \mid C \cap \bar{E})P(C \mid \bar{E})P(\bar{E})$$
$$= \frac{NBS}{NBIT} \cdot \frac{1}{NGB} \cdot \left[1 - \frac{NDS}{ND}\right]$$

Therefore,

$$P(B \cap C) = \frac{1}{NGB}\left\{NGB \cdot \frac{NDS_G}{ND} + \frac{NBS}{NBIT}\left[1 - \frac{NDS}{ND}\right]\right\} \tag{A.65}$$

From Propositions 1 and 3

$$NBS = NBIT\left[1 - e^{-\frac{NDS_G}{NBIT}}\right] \tag{A.66}$$

and

$$NDS = ND\left[1 - e^{-\frac{NTS}{ND}}\right] \tag{A.67}$$

By combining Equations A.53 and A.64 through A.67, the following expression for LTG will result:

$$LTG = \frac{1}{NGB}\left[1 - e^{-\frac{NTS}{ND}} e^{-\frac{NDS_G}{NBIT}}\right] \tag{A.68}$$

where NDS_G is computed using Equations A.53 and A.67.

(c) Deriviation of ENC(1) for the BBH-TPF Equi-Join Algorithm

Let

A,E be as defined in Section A.2.a and

B be as defined in section A.3.a

The quantity ENCT(BBH-TPF) can be expressed as follows:

$$ENCT(BBH-TPF) = NTS_{PV} \cdot NDS_B \cdot P(A \cap B) \tag{A.69}$$

$$\begin{aligned} P(A \cap B) &= P(A \cap B \cap E) + P(A \cap B \cap \bar{E}) \\ P(A \cap B \cap E) &= P(A \mid B \cap E)P(B \mid E)P(E) \\ &= 1 \cdot 1 \cdot \frac{NDSp}{ND} = \frac{NDSp}{ND} \\ P(A \cap B \cap \bar{E}) &= P(A \mid B \cap \bar{E})P(B \mid \bar{E})P(\bar{E}) \end{aligned}$$

and

$$= \frac{NBF}{NBP} \cdot \frac{NBS}{NBIT}\left[1 - \frac{NDSp}{ND}\right]$$

Therefore

$$P(A \cap B) = \frac{NDSp}{ND} + \frac{NBS}{NBIT} \cdot \frac{NBF}{NBP}\left[1 - \frac{NDSp}{ND}\right] \tag{A.70}$$

Also ENC(1) of the BBH-TPF equi-join algorithm can be expressed as follows:

$$ENC(1) = NTT \cdot ENCT(BBH-TPF) \tag{A.71}$$

By combing Equations A.69 through A.71, the following formula will result.

$$ENC(1) = NTT \cdot NTS_{PV} \cdot NDS_B\left\{\frac{NDSp}{ND} + \frac{NBS}{NBIT} \cdot \frac{NBF}{NBP}\left[1 - \frac{NDSp}{ND}\right]\right\} \tag{A.72}$$

Using Proposition 3, it can be shown that the following equation is true for the BBH-TPF equi-join algorithm.

$$NBS = NBIT\left[1 - e^{-\frac{NDS_M}{NBIT}}\right] \tag{A.73}$$

By combining Equations A.15 through A.19, A.27 and A.30 together with A.72 and A.73), the following expression for ENC(1) will result:

$$ENC(1) = \frac{NTT \cdot NTS}{NMS \cdot NP \cdot NBP}\left\{\frac{1 - e^{-\frac{NTSp}{ND}} + \left[1 - e^{-\frac{NDS_M}{NBIT}}\right]\left[1 - e^{-\frac{NDS_p}{NBP}}\right]e^{-\frac{NTS_p}{ND}}}{1 - e^{-\frac{NDSp}{NBP}}}\right\} \tag{A.74}$$

(d) Deriviation of ENC(1) for the BHH-TPF Equi-Join Algorithm

Let

A,D and E be as defined in Section A.2.b and

B be as defined in Section A.3.a.

The quantity ENCT(BHH-TPF) can be expressed as follows:

$$ENCT(BHH-TPF) = NTS_{MV} \cdot NDS_B \cdot P(A \cap B \cap D) \tag{A.75}$$

$$P(A \cap B \cap D) = P(A \cap B \cap D \cap E) + P(A \cap B \cap D \cap \bar{E})$$
$$P(A \cap B \cap D \cap E) = P(A \mid B \cap D \cap E) P(D \mid B \cap \bar{E}) P(B \mid \bar{E}) P(\bar{E})$$
$$= 1 \cdot \frac{NDS_P}{NDS_M} \cdot 1 \cdot \frac{NDS_M}{ND} = \frac{NDS_P}{ND}$$

and

$$P(A \cap B \cap D \cap \bar{E}) = P(A \mid B \cap D \cap \bar{E}) P(D \mid B \cap \bar{E}) P(B \mid \bar{E}) P(\bar{E})$$
$$= \frac{NBF}{NBP} \cdot \frac{1}{NP} \cdot \cdot \frac{NBS}{NBIT} \cdot \left[1 - \frac{NDS_M}{ND}\right]$$

Therefore,

$$P(A \cap B \cap D) = \frac{1}{NP}\left\{NP \cdot \frac{NDS_P}{ND} + \frac{NBF}{NBP} \cdot \frac{NBS}{NBIT}\left[1 - \frac{NDS_M}{ND}\right]\right\} \tag{A.76}$$

By combining Equations A.23 and A.76, the following formula for $P(A \cap B \cap D)$ will result:

$$P(A \cap B \cap D) = \frac{1}{NP}\left\{\frac{NDS_M}{ND} + \frac{NBF}{NBP} \cdot \frac{NBS}{NBIT}\left[1 - \frac{NDS_M}{ND}\right]\right\} \tag{A.77}$$

Also ENC(1) of the BHH-TPF equi-join algorithm can be expressed as follows:

$$ENC(1) = NTT \cdot ENCT(BHH-TPF) \tag{A.78}$$

By combining Equations A.75, A.77 and A.78, the following formula will result:

$$ENC(1) = NTT \cdot NTS_{MV} \cdot NDS_B \cdot \frac{1}{NP}\left\{\frac{NDS_M}{ND} + \frac{NBF}{NBP} \cdot \frac{NBS}{NBIT}\left[1 - \frac{NDS_M}{ND}\right]\right\}$$

(A.79)

By combining Equations A.27 through A.31 and A.73 together with A.69, the following expression for ENC(1) will result:

$$ENC(1) = \frac{NTT \cdot NTS}{NMS \cdot NP^2 \cdot NBP}\left\{\frac{1 - e^{-\frac{NTS_M}{ND}} + \left[1 - e^{-\frac{NDS_P}{NBP}}\right]\left[1 - e^{-\frac{NDS_M}{NBIT}}\right]e^{-\frac{NTS_M}{ND}}}{1 - e^{-\frac{NDSp}{NBP}}}\right\}$$

A.80)

(e) Deriviation of ENC(1) for the HBH-TPF Equi-Join Algorithm

Let

A,C and E be as defined in Section A.2.c and

B be as defined in Section A.3.b

The quantity ENCT(HBH-TPF) can be expressed as follows:

$$ENCT(HBH-TPF) = NTS_{PV} \cdot NDS_B P(A \cap B \cap C) \qquad (A.81)$$

$$P(A \cap B \cap C) = P(A \cap B \cap C \cap E) + P(A \cap B \cap C \cap \bar{E})$$

$$P(A \cap B \cap C \cap E) = P(A \mid B \cap C \cap E)P(B \mid C \cap E)P(C \mid E)P(E)$$

$$= 1 \cdot 1 \cdot \frac{NDSp}{NDS_S} \cdot \frac{NDS_S}{ND} = \frac{NDSp}{ND}$$

and

$$P(A \cap B \cap C \cap \bar{E}) = P(A \mid B \cap C \cap \bar{E})P(B \mid C \cap \bar{E})P(C \mid \bar{E})P(\bar{E})$$

$$= \frac{NBF}{NBP} \cdot \frac{NBS}{NBIT} \cdot \frac{1}{NGB} \cdot \left[1 - \frac{NDS_S}{ND}\right]$$

Therefore,

$$P(A \cap B \cap C) = \frac{1}{NGB}\left\{NGB \cdot \frac{NDS_P}{ND} + \frac{NBF}{NBP} \cdot \frac{NBS}{NBIT}\left[1 - \frac{NDS_S}{ND}\right]\right\} \qquad (A.82)$$

By combining Equations A.35 and A.82, the following formula for $P(A \cap B \cap C)$ will result:

$$P(A \cap B \cap C) = \frac{1}{NGB}\left\{\frac{NDS_S}{ND} + \frac{NBF}{NBP} \cdot \frac{NBS}{NBIT}\left[1 - \frac{NDS_S}{ND}\right]\right\} \qquad (A.83)$$

It is easy also to see that

$$ENC(1) = NTT \cdot ENCT(HBH - TPF) \qquad (A.84)$$

By combining Equations A.81, A.83 and A.84, the following formula will result:

$$ENC(1) = NTT \cdot NTS_P \cdot NDS_B \frac{1}{NGB}\left\{\frac{NDS_S}{ND} + \frac{NBF}{NBP} \cdot \frac{NBS}{NBIT}\left[1 - \frac{NDS_S}{ND}\right]\right\} \qquad (A.85)$$

Using Proposition 3, it can be shown that the following equation is true for the HBH-TPF equi-join algorithm:

$$NBS = NBIT\left[1 - e^{-\frac{NDS_G}{NBIT}}\right] \qquad (A.86)$$

By combining equations A.39 through A.42, A.44, A.53 and A.85, the following expression for ENC(1) will result:

$$ENC(1) = \frac{NTT \cdot NTS}{NGB^2 \cdot NP \cdot NBP}\left\{\frac{1 - e^{-\frac{NTS_S}{ND}} + \left[1 - e^{-\frac{NDS_P}{NBP}}\right]\left[1 - e^{-\frac{NDS_G}{NBIT}}\right] e^{-\frac{NTS_S}{ND}}}{1 - e^{-\frac{NDS_P}{NBP}}}\right\}$$

(A.87)

(f) Deriviation of ENC(1) for the HHH-TPF Equi-Join Algorithm

Let

A,D, C and E be as defined in Section A.2.d and

B be as defined in Section A.3.b

The quantity ENCT(HHH-TPF) can be expressed as follows:

$$ENCT(HHH-TPF) = NTS_{CV} \cdot NDS_B \cdot P(A \cap B \cap C \cap D) \qquad (A.88)$$

$$P(A \cap B \cap C \cap D) = P(A \cap B \cap C \cap D \cap E) + P(A \cap B \cap C \cap D \cap \bar{E})$$

$$P(A \cap B \cap C \cap D \cap E) = P(A \mid B \cap C \cap D \cap E)P(D \mid B \cap C \cap D \cap E)P(B \mid C \cap E)P(C \mid E)P(E)$$

$$= 1 \cdot \frac{NDS_P}{NDS_G} \cdot 1 \cdot \frac{NDS_G}{NDS} \cdot \frac{NDS}{ND} = \frac{NDS_P}{ND}$$

and

$$P(A \cap B \cap C \cap D \cap \bar{E}) = P(A \mid B \cap C \cap D \cap \bar{E})P(D \mid B \cap C \cap \bar{E})P(B \mid C \cap \bar{E})P(C \mid \bar{E})P(\bar{E})$$

$$= \frac{NBF}{NBP} \cdot \frac{1}{NP} \cdot \frac{NBS}{NBIT} \cdot \frac{1}{NGB} \cdot \left[1 - \frac{NDS}{ND}\right]$$

Therefore,

$$P(A \cap B \cap C \cap D) = \frac{1}{NP \cdot NGB}\left\{NP \cdot NGB \cdot \frac{NDS_P}{ND} + \frac{NBF}{NBP} \cdot \frac{NBS}{NBIT} \cdot \left[1 - \frac{NDS}{ND}\right]\right\}$$

(A.89)

By combining Equations A.53, A.55 and A.89, the following equation will result:

$$P(A \cap B \cap C \cap D) = \frac{1}{NP \cdot NGB}\left\{\frac{NDS}{ND} + \frac{NBF}{NBP} \cdot \frac{NBS}{NBIT}\left[1 - \frac{NDS}{ND}\right]\right\} \qquad (A.90)$$

It is also easy to see that

$$ENC(1) = NTT \cdot ENCT(HHH-TPF) \qquad \textbf{(A.91)}$$

By combining Equations A.88, A.90 and A.91, the following equation will result:

$$ENCT(1) = NTT \cdot NTS_{GV} \cdot NDS_B \cdot \frac{1}{NP \cdot NGB}\left\{\frac{NDS}{ND} + \frac{NBF}{NBP}\cdot\frac{NBS}{NBIT}\left[1-\frac{NDS}{ND}\right]\right\}$$

(A.92)

By combining Equations A.52 through A.58, (A.66), and A.92, the following equation for ENC(1) will result:

$$ENCT(1) = \frac{NTT \cdot NTS}{NGB^2 \cdot NP^2 \cdot NBP}\left\{\frac{1-e^{-\frac{NTS}{ND}} + \left[1-e^{-\frac{NDS_B}{NBP}}\right]\left[1-e^{-\frac{NDS_G}{NBIT}}\right]e^{-\frac{NTS}{ND}}}{1-e^{-\frac{NDS_B}{NBP}}}\right\}$$

(A.93)

1.5. Derivations for the STPF Equi-Join Execution Models

In this section, the derivation of the ENC(1) for both the HBH-STPF and the HHH-STPF equi-join algorithms is presented.

(a) Deriviation of ENC(1) for the HBH-STPF Equi-Join Algorithm

Let

A,C and E be as defined in Section A.2.c and

B be as defined in Section A.3.b.

The quantity ENCT(HBH-STPF) can be expressed as follows:

$$ENCT(HBH-STPF) = NTS_V \cdot NDSF_B \cdot P(A \cap B \cap C) \qquad (A.94)$$

where $NDSF_B$ is the number of distinguished values in the join attribute of the source tuples which survive the BIT-T checking and are stored in one hash-

table bucket.

Following the steps used in deriving equation A.83 which computes the quantity $P(A \cap B \cap C)$ for the HBH-TPF algorithm, it can be shown that $P(A \cap B \cap C)$ of Equation A.94 can be expressed as follows:

$$P(A \cap B \cap C) = \frac{1}{NGB}\left\{\frac{NDS_S}{ND} + \frac{NBF}{NBP}\cdot\frac{NBS_S}{NBIT}\left[1-\frac{NDS_S}{ND}\right]\right\} \qquad (A.95)$$

It is easy to see that

$$ENC(1) = NTT \cdot ENCT(HBH-STPF) \qquad (A.96)$$

By combining Equations A.94 through A.96, the following equation for ENC(1) will result:

$$ENC(1) = \frac{NTT \cdot NTS_{PV} \cdot NDSF_B}{NGB}\left\{\frac{NDS_S}{ND} + \frac{NBF}{NBP}\cdot\frac{NBS_S}{NBIT}\left[1-\frac{NDS_S}{ND}\right]\right\} \qquad (A.97)$$

For an STPF algorithm, the following equations are true:

$$NBF = NBP\left[1-e^{-\frac{NDSF_p}{NBP}}\right] \qquad (A.98)$$

and

$$NDSF_B = \frac{NDSF_p}{NBF} \qquad (A.99)$$

where *NDSFp* is the number of distinguished values in the join attribute of the source tuples of one triplet which survive the BIT-T checking.

A formula for *NDSFp* can be derived as follows:

Let

B be the event that a source value hashes to a "set" bit in the BIT-T and

C and E be as defined in Section A.3.b.

For the HBS-STPF algorithm, $NDSF_p$ can be expressed as follows:

$$NDSF_p = NDS_S \cdot P(B \cap C) \tag{A.100}$$

$$\begin{aligned} P(B \cap C) &= P(B \cap C \cap E) + P(B \cap C \cap \bar{E}) \\ P(B \cap C \cap E) &= P(B \mid C \cap E) P(C \mid E) P(E) \\ &= 1 \cdot \frac{NDT_G}{NDT} \cdot \frac{NDT}{ND} = \frac{NDT_G}{ND} \\ P(B \cap C \cap \bar{E}) &= P(B \mid C \cap \bar{E}) P(C \mid \bar{E}) P(\bar{E}) \end{aligned}$$

and

$$= \frac{NBS_T}{NBIT} \cdot \frac{1}{NGB} \left[1 - \frac{NDT}{ND} \right]$$

Therefore,

$$P(B \cap C) = \frac{1}{NGB} \left\{ NGB \cdot \frac{NDT_G}{ND} + \frac{NBS_T}{NBIT} \left[1 - \frac{NDT}{ND} \right] \right\} \tag{A.101}$$

For a " global hash" algorithm,

$$NDT = NGB \cdot NDT_G \tag{A.102}$$

By combining Equations A.100 through A.102, the following expression for $NDSF_p$ will result:

$$NDSF_p = NDS_S \cdot \frac{1}{NGB} \left\{ \frac{NDT}{ND} + \frac{NBS_T}{NBIT} \left[1 - \frac{NDT}{ND} \right] \right\} \tag{A.103}$$

Since

$$NDS_P = \frac{NDS_S}{NGB}$$

and

$$LS = \frac{1}{NGB}\left\{\frac{NDT}{ND} + \frac{NBS_T}{NBIT}\left[1-\frac{NDT}{ND}\right]\right\}$$

then

$$NDSF_P = NGB \cdot LS \cdot NDSp \tag{A.104}$$

By combining Equations A.39 through A.43, A.65, A.97 through A.99 and A.104, the following equation for ENC(1) will result:

$$ENC(1) = \frac{NTT \cdot NTS \cdot LS}{NP \cdot NGB \cdot NBP}\left\{\frac{1-e^{-\frac{NTS_S}{ND}} + e^{-\frac{NTS_S}{ND}}\left[1-e^{-\frac{NDSF_p}{NBP}}\right]\left[1-e^{-\frac{NDS_G}{NBIT}}\right]}{1-e^{-\frac{NDSFp}{NBP}}}\right\}$$

(A.105)

(b) Deriviation of ENC(1) for the HHH-STPF Equi-Join Algorithm

Let

A,D,C

be as defined in Section A.2.d and

B as defined in Section A.3.b.

The quantity ENCT(HHH-STPF) can be expressed as follows:

$$ENCT(HHH-STPF) = NTS_{GV} \cdot NDSF_B \cdot P(A \cap B \cap C \cap D) \tag{A.106}$$

Following the steps used in deriving Equation A.90 which computes the quantity $P(A \cap B \cap C \cap D)$ for the HHH-TPF algorithm, it can be shown that $P(A \cap B \cap C \cap D)$ of Equation A.106 can be expressed as follows:

$$P(A \cap B \cap C \cap D) = \frac{1}{NP \cdot NGB}\left\{\frac{NDS}{ND} + \frac{NBF}{NBP} \cdot \frac{NBS_S}{NBIT}\left[1-\frac{NDS}{ND}\right]\right\} \tag{A.107}$$

It is easy to see that

$$ENC(1) = NTT \cdot ENCT(HHH-STPF) \tag{A.108}$$

By combining Equations A.106 through A.107, the following equation for $ENC(1)$ will result:

$$ENC(1) = \frac{NTT \cdot NTS_{GV} \cdot NDSF_B}{NP \cdot NGB}\left\{\frac{NDS}{ND} + \frac{NBF}{NBP} \cdot \frac{NBS_S}{NBIT}\left[1-\frac{NDS}{ND}\right]\right\} \tag{A.109}$$

where

$$NDSF_B = \frac{NDSFp}{NBF} \tag{A.110}$$

A formula for $NDSFp$ can be derived as follows:

For a "local hash" STPF algorithm, $NDSF_P$ can be expressed as follows:

$$NDSF_p = NDSp \cdot (B \cap C \cap D)$$

where

B is defined as in Section A.4.2 and

C,D are defined as in Section A.3.f

By following the same steps of the derivation for $NDSFp$ of HBH-STPF algorithm, the following formula for $NDSF_p$ of the HHH-STPF algorithm will result:

$$NDSF_p = \frac{NDS}{NP \cdot NGB}\left\{\frac{NDT}{ND} + \frac{NBS_T}{NBIT}\left[1-\frac{NDT}{ND}\right]\right\}$$

For an STPF algorithm,

$$LS = \frac{1}{NGB}\left\{\frac{NDT}{ND} + \frac{NBST}{NBIT} \cdot \left[1-\frac{NDT}{ND}\right]\right\}$$

and

$$NDS_P = \frac{NDS}{NP \cdot NGB}$$

Therefore,

$$NDSF_P = NGB \cdot LS \cdot NDS_P \qquad \text{(A.111)}$$

By combining Equations A.52 through A.55, A.57, A.97, A.98, A.108 through A.110 and A.111, the following equation for ENC(1) will result:

$$ENC(1) = \frac{NTT \cdot NTS \cdot LS}{NP^2 \cdot NGB \cdot NBP} \left\{ \frac{1 - e^{-\frac{NTS}{ND}} + e^{-\frac{NTS}{ND}} \left[1 - e^{-\frac{NDSF_p}{NBP}}\right] \left[1 - e^{-\frac{NDS_G}{NBP}}\right]}{1 - e^{-\frac{NDSF_p}{NBP}}} \right\}$$

(A.112)

1.6. Derivations for the STCF Equi-Join Execution Models

In this section, the derivations of the formulas which compute the parameters LT and LS of the STCF execution models are presented.

(a) Derivation of the LT Formula

Let

B be the event that a target tuple will hash to a "set" bit in BIT-S and

E be the event that a target tuple's join attribute value has a match among those of the source tuples.

Then

$$LT = P(B) \qquad \text{(A.113)}$$

In general P(B) can be computed from the following formula:

$$P(B) = P(B \cap E) + P(B \cap \bar{E})$$
$$= P(B \mid E)P(E) + P(B \mid \bar{E})P(\bar{E})$$

Therefore,

$$P(B) = 1 \cdot \frac{NDS}{ND} + \frac{NBS}{NBIT}\left[1 - \frac{NDS}{ND}\right] \qquad \text{(A.114)}$$

Using Proposition 1 and 3, it can be shown that the following equations are true for the STCF execution models:

$$NBS = NBIT\left[1 - e^{-\frac{NDS}{NBIT}}\right] \qquad \text{(A.115)}$$

and

$$NDS = ND\left[1 - e^{\frac{-NTS}{ND}}\right] \qquad \text{(A.116)}$$

Combining Equations A.113 through A.116, the following formula for LT will result:

$$LT = 1 - e^{\frac{-NTS}{ND}} e^{-\frac{NTS}{ND}} \qquad \text{(A.117)}$$

(b) Derivation of the LS Formula

Let

B be the event that a source tuple will hash to a "set" bit in BIT-T and

E be the event that a source tuple's join attribute value has a match among those of the target tuples.

By following the same steps used in the derivation of the LT formula, it can be shown that

$$LS = 1 - e^{-\frac{NTT}{ND}} e^{-\frac{NDT}{NBIT}} \qquad \text{(A.118)}$$

where NDT is the number of distinguished values in the Join attribute of the target tuples.

Appendix B

Values For The Parameters of The Equi-Join Execution Models

In general the parameters of the equi-join execution models can be grouped into two categories, namely, the static parameters and the dynamic parameters. A parameter is static/dynamic if its value (stays the same)/(changes) throughout the study of the models' output parameters behavior.

In this appendix, a value for every static parameter is presented. A range of values for every dynamic parameter is also presented.

1.1. Values For The Static Parameters

In Table B.1 the static parameters as well as their values are displayed. The values of the hardware-disk parameters are those of an IBM 3380 moving-head disk. The disk is assumed to be modified for the parallel read/write of a whole cylinder. It is also assumed that this modification does not affect the disk transfer rate. The time to read/write one track of the disk is taken as the value of the time to read/write the whole cylinder (TDT). It is also assumed that the value of the parameter TBT is the same as that of TDT.

A cylinder of an IBM 3380 moving-head disk [IBM80] consists of 15 tracks. Thus the number of triplets within an associative cluster is 15. Also the MAUC is equal to the capacity of the 15 tracks. It is also assumed that BUFS of the PC's triplets has the same capacity as that of an MAU (cylinder). Thus the value of BUFSC is the same as that of MAUC.

Table B.1
values for the Static Parameters of the Equi-Join Execution Models

Parameter Type	Parameter Name	Parameter Value	Parameter Unit
Data	LTS LTT LJ	100 100 10	Byte Byte Byte
Hardware-Disk	MAUC TDAC TSK TDT	$.71\times10^{+6}$ 16 3 16.7	Byte msec msec msec
Hardware-PC-PB	TBT	16.7	msec
Hardware-PC	NP BUFSC	15 $.71\times10^{+6}$	 Byte
Hardware-Triplet	TCD TCI TH TMS TMT TMO TEP	.02 .021 .102 .091 .091 .179 .007	msec msec msec msec msec msec msec
Hardware-Master	TS TT	.1 .1	msec msec

In calculating the values of the hardware-triplet parameters, It is assumed that the triplet processor is an Intel 8086/1 microprocessor [MAN86]. In general, a hardware-triplet parameter represents an operation carried out by the corresponding microprocessor. To calculate the value of a such parameter, a procedure was written, in the 8086 assembly language, which carries out the corresponding operation. The execution times, obtained from the Intel 8086 microprocessor user's manual [MAN86], for the instructions in the

procedure were added to obtain the value of the corresponding parameter.

```
;This section of code compares, directly, the join attribute
;value of two source and target tuples.
;
;The following are assumed:
;
;     JAT_PTR     Points to the join attribute of the current target tuple

;     BX          Points to the join attribute of the previous source tuple

;     ES_PTR      Points to the end of the source tuples.
;
      15     TCD:   ADD          BX,LJ          ;BX points to current S tuple
      15            CMP          BX,ES_PTR      ;End of source tuples ?
       4            JGE          OUT            ;Jump if yes
;
       2            MOV          SI,BX.         ;SI points to the join attribute
;                                                 of the current source tuple
      14            MOV          DI,JAT_PTR     ;DI points to the join
;                                                 of the current target tuple
      14            MOV          CX,LJ          ;CX gets the length of the join
;                                                 attribute
;
     119            REP CMPS     WORDPTR
      16            JNE          TCD
;
                OUT:                           ;Move to process next T tuple
```

Procedure B.1 A Procedure for Calculating the Parameter TCD

The procedures B.1, B.2, B.4, B.5 and B.6 are developed for calculating the values of the parameters, TCD, TCI, TMS or TMT, TEP and TMO, respectively. The integer written beside each instruction within each of these procedures indicates the number of clock cycles needed to execute the corresponding instruction. by adding the clock cycles of each instruction within a given procedure and multiplying the result by the cycle time of an 8086/1 microprocessor (.1 us), the value of the corresponding parameter is obtained.

```
;This section of code compares, indirectly, the join attribute
;value of two source and target tuples
;
;The following are assumed:
;
;    JAT_PTR     Points to the join attribute of the current target tuple
;    BX          Points to the address field of the previous source tuple
;    JA_DISP     Displacement (in bytes) of the join attribute from
;                the start of the tuple
;
      13   TCI:  mov        SI,[BX]     ;SI point to the current source tuple
       4         CMP        SI,0        ;Is it last source tuple in link list
       4         JE         OUT         ;Jump if yes
;
       2         MOV        BX,SI
      15         ADD        SI,JA_DISP  ;Advance SI to next tuple
      14         MOV        DI,JAT_PTR
      14         MOV        CX,LJ
;
     119         REPE CMPS  WORDPTR
      16         JNE        TCI
;
     OUT:                               ;Proceed to process the next target tuple
```

Procedure B.2 A Procedure for Calculating the Parameter TCI

```
;This section of code calculate a typical hashing function.
;The input to the function is the Join attribute value.
;
;The following have been assumed:
;
;     SI          Points to the tuple bieng hashed
;     JA_DISP       displacement(in bytes) of the Join
;                 Attribute from the start of the tuple
;     MUL_FACTOR    The hashing function multiplication factor
;     DIV_FACTOR    The hashing function division factor
;
          3     TH      SUB     AX,AX           ;Clear the accumulator
         15             ADD     SI,JA_DISP      ;SI points to the Join attribute
         14             MOV     CX,LJ           ;CX is a counter
     ;
         70     LADD:   ADD     AX,[SI]
         20             ADD     SI,2            ;update SI
         85             LOOP    LADD
     ;
        138             MUL     MUL_FACTOR
        165             DIV     DIV_FACTOR      ;Quotient in AX, Remainder in DX
     ;
```

Procedure B.3 A Procedure for Calculating the Parameter TH

```
;This section of code moves a source or target tuple ;within the triplet's local
memory unit
;
;The following have been assumed:
;
;     SI        Points to the tuple to be transfered
;     LT        Tuple length (100 bytes)
;     BUF_PTR   Pointer to the destination buffer
;
          14    TM:   MOV        DI,BUF_PTR      :DI points to destination buffer
          14          MOV        CX,LT           ;Load counter
         859          REP MOVS   ;Move the tuple
          15          MOV        BUF_PTR,DI      ;Update BUF_PTR
     ;
```

Procedure B.4 A Procedure for calculating the Parameter TM

```
;This section of code exchange the content of two location,
;each of two bytes long, within the triplet LMU.
;
;The following have been assumed:
;
;     BX      Points to one of the locations
;     PTR     Points to the second location
;
          13      TEP:      MOV      AX,[BX]      ;AX gets the content of first locatio
          14                MOV      SI,PTR       ;SI points to second location
          13                MOV      DX,[SI]      ;DX gets the content second locatio
          14                MOV      [SI],AX      ;Exchange
          14                MOV      [BX],DX
```

Procedure B.5 A Procedure for calculating the Parameter TEP

```
;This section of code moves an output tuple to the output buffer
;
;The following have been assumed:
;
;     BX        BX points to the source tuple
;     LTS       Source tuple length
;     T_PTR      Pointer to target tuple
;     LTT       Target tuple length
;     BUF_PTR    Pointer to the output buffer
;
          2      TMO:   MOV          SI,BX          ;SI points to source tuple
         14             MOV          DI,BUF_PTR     ;DI points to output buffer
         14             MOV          CX,LTS
        859             REP MOVSW                   ;Move the source tuple
     ;
         14             MOV          SI,T_PTR       ;SI points to target tuple
         14             MOV          CX,LTT
        859             REP          MOVSW          ;Move the target tuple
     ;
         15             MOV          BUF_PTR,DI     ;Update BUF_PTR
```

Procedure B.6 A Procedure for Caculating the Parameter TMO

In general, the proposed RDBM will have a family of hashing functions, each is suitable for a class of the join attributes. Each hashing function will implemented as a subroutine. Figure B.3 shows a procedure which implements a typical hashing function. This function was used to calculate the parameter TH. In addition, the latter value of TH was multiplied by a factor of two to account for the overhead time associated with the subroutine implementation of the hashing function.

The values of the hardware-master parameters (TS or TT) are dependent on the implementation of the master/triplets communication structure (software + hardware). Since the details of such implementation are not available, TS and TT are calculated assuming that the TBUS has an effective bandwidth of 1Mbyte/s. This is a reasonable choice since such bus is within the current technology limits.

1.2. Values for the Dynamic Parameters

The dynamic parameters can be grouped into two categories. The first category includes the parameters NTS, NTT and ND. The second category includes the parameters NBP and NBIT. The parameters of the first category depend only on the data participating in the equi-join operation. Thus the selection of values for these parameters are independent of the specific execution model they characterize.

In this performance study of the equi-join algorithms, the parameters NTS and NTT have equal values. This is done in order to reduce the number of parameters which need to be varied throughout the performance investigation. The parameter NTS(NTT) is varied between the limits 10^3 and 10^5. The parameter ND is not assigned values directly, but rather indirectly through assigning values to the quantity NTS/ND (NTT/ND). This is possible since when-

ever the parameter ND appears in a formula of those generated for the equi-join execution models, it will be in the form (NTS/ND) or (NTT/ND). To understand the meaning of the ratio NTS/ND, recall from Appendix B that the probability P of a target tuple finds at least one source tuple with matching join attribute value (NDS/ND). Therefore

$$P = \frac{NDS}{ND} = 1 - e^{-\frac{NTS}{ND}}$$

For small $\frac{NTS}{ND}$ (<.1) the expression $1 - e^{-\frac{NTS}{ND}}$ can be approximated by NTS/ND. That is, for small NTS/ND the ratio $\frac{NTS}{ND}$ can be interpreted as the probability that a target tuple will find at least one source tuple with matching join attribute value.

The performance of the equi-join algorithms will be evaluated when the ratio $NTS/ND(NTT/ND)$ has the values .01, .1 and 1. While the values (.01) and (.1) represent the limits of a range of values where P is small, the values (.1) and (1) represent the limits of a range of values where P is relatively large.

In the following, the values that both NBIT and NBP will have throughout the equi-join performance evaluation are presented.

1.2.1. Values for NBP

Two factors put an upper limit on the range of values that the parameter NBP can take. These two factors are:

(1) The Domain Limitation

A hashing function F can be defined as the mapping

$$F: D \rightarrow R$$

where D is the domain of F and R is the range of F.

The mapping function F must be an "onto" one. That is, for every $r \in R$, there exists at least one $d \in D$ which is mapped by F or r.

The above condition (the "Onto" condition) puts an upper limit on the range of values that NBP can take. Recall that NBP is the range of the "hash table" hashing function. Let D be the domain of the latter function (D has the cardinality NDD). Then NBP must be chosen such that the "onto" condition is satisfied. That is, NBP must be smaller than NDD, the cardinality of the hashing function's domain.

Throughout this study of equi-join algorithm performance it is assumed that

$$NBP \leq \frac{NDD}{4} \quad .$$

For the algorithms BBH-Basic, BBH-TPF and BBH-STPF, NDD is the same as ND.

For the algorithms BHH-Basic, BHH-TPF and BHH-STPF, NDD is ND/NP. This is because the local hashing method partitions, on the average, the Join attribute underlying domain into NP disjoint sections.

For the algorithms HBH-Basic, HBH-TPF and HBH-STPF, NDD is ND/NGB. This is because the global hashing method partitions, on the average, the join attribute underlying domain into NGB disjoint sections .

For the algorithms HHH-Basic, HHH-TPF and HHH-STPF, NDD is $ND/NGB \cdot NP$. This is because first the global hashing method partitions the join attribute underlying domain into NGB disjoint sections then each section is partioned further, by the local hashing method, into NP disjoint subsections.

(2) The Storage Limitation

Recall that the hash table of a triplet is organized as two separate areas, namely, the primary and the secondary ones. The secondary area stores the source tuples. The primary area is divided into buckets each is capable of storing a pointer (has a size of 2 bytes) to the link list which stores all the source tuples which hash to the corresponding bucket. The number of buckets (NBP) is limited by the capacity of storage allocated to the primary area.

Let FNBP be the ratio of the upper limit on NBP (SUNBP), due to storage limitation, and the average number of source tuples in one triplet.

For the "global broadcast" basic and TPF algorithms, the SUNBP is computed from the following formula:

$$SUNBP \doteq \begin{cases} \left\lfloor FNBP \cdot \dfrac{NTS}{NMS \cdot NP} \right\rfloor & \text{for } \textit{the global Broadcast Basic} \text{ and } \textit{TPF algorithms} \\ \left\lfloor FNBP \cdot \dfrac{NTS}{NGB \cdot NP} \right\rfloor & \text{for } \textit{the STPF} \text{ and } \textit{the global hash basic} \text{ and } \textit{TPF algorithms} \end{cases}$$

Recall that throughout this performance study, It is assumed that BUFSC be equal to MAUC. This results in both the quantities NMS and NGB having equal values. Therefore, the quantity UNBP for all the equi-join algorithm can be expressed as follows:

$$SUNBP = \left\lfloor FNBP \cdot \frac{NTS}{NMS \cdot NP} \right\rfloor$$

Multiplying *SUNBP* of the above equation by two (the bucket size) gives the storage capacity which must be allocated to the primary area for the given algorithm. By varying the parameter FNBP, the effect of the primary area storage capacity on the performance of the equi-join algorithms will be studied.

Throughout the equi-join performance analysis, NBP will be chosen as the highest value which does not violate both the domain and the storage limits. Therefore,

$$NBP = MIN \left\{ \frac{NDD}{4} , SUNBP \right\}$$

1.2.2. Values for NBIT

Two factors put an upper limit on the range of values that the parameter NBIT can take. These two factors are:

(1)The Domain Limitation

Recall that NBIT is the range of the "hash-bit" hashing function. Following the discussion of the previous subsection it can be shown that NBIT must fulfill the following inequality;

$$NBIT \leq \frac{NDD}{4}$$

where

$$NDD = \begin{cases} ND & \text{for } \textit{the global broadcast with TPF} \\ & \text{and } \textit{STCF algorithms} \\ (ND / NGB) & \text{for } \textit{the global hash with TPF} \\ & \text{and for } \textit{STPF algorithms} \end{cases}$$

(2) The Storage Limitation

Recall that the 1-bit vector(s) is initialized and maintained by the cluster master processor. The storage capacity allocated for the implementation of the 1-bit vectors puts an upper limit on the number of bits the vector(s) will have.

Let FNBIT be the ratio of the upper limit on NBIT, due to the storage limitations and the sum of NTS and NTT.

Then

$$SUNBIT = FNBIT \cdot (NTS + NTT)$$

Thus NBIT must fulfill the following inequality:

$$NBIT \leq \begin{cases} SUNBIT & \text{for } \textit{all algorithms with one vector} \\ (SUNBIT/2) & \text{for } \textit{all algorithms with two vectors.} \end{cases}$$

The effect of the storage limitation on the models' output parameters will be studied through varying the parameter FNBIT.

Throughout the equi-join performance analysis, NBIT will be chosen as the highest value which does not violate both the domain and the storage limits. Therefore,

$$NBIT = \begin{cases} MIN\left\{(NDD/4),\ SUNBIT\right\} & \text{for } \textit{all algorithms with one vector} \\ MIN\left\{(NDD/4),\ (SUNBIT/2)\right\} & \text{for } \textit{all algorithms with two vector.} \end{cases}$$

BIBLIOGRAPHY

BIBLIOGRAPHY

[BABB79] E. Babb, "Implementing a Relational Database by Means of Specialized Hardware," ACM Trans. Database System, Vol. 4, No. 1 (March 1979), pp. 1-29.

[BANC80] F.Bancihon and M. Scholl, "Design of a Backend Processor for a Database Machine," Proc. of the ACM SIGMOD, 1980, International Conference of Mangement of Data, (May 1980).

[BANE79] J. Banerjee, D. Hsiao and K. Kannan, "DBC-A Database Computer for Very Large Databases," IEEE Transaction on Computers, Vol. C-28, No. 6, June 1979, pp. 414-429.

[BAUM76] R. Baum, D. Hsiao and K. Kannan, "The Architecture of a Database Computer, Part I: Concepts and Capabilities," OSU-CISRC-TR-76-1, OHIO State University, Sept. 1976.

[BORA81] H. Boral, "On the Use of Data-Flow Techniques In Database Machines," Ph.D. Thesis, 1981. The Computer Sciences Department, University of Wisconsin, Madison.

[CANA74] R. H. Canaday, et al., "A Backend Computer for Database Mangement," CACM, Oct. 1974, Vol. 17, No. 10.

[CHAM76] D. D. Chamberlin, "Relational Database Management System," Computing Surveys, Vol. 8, No. 1, March 1976.

[CHEN78] T.C. Chen and H. Chang, "Magnetic Bubble Memory and Logic," Advances in Computers, Vol. 17, 1978, pp. 228-279.

[CODD72] E. F. Codd, "Relational Completeness of Database Sublanguages," Database Systems, Courant Computer Science Symposia, Vol. 6, Englewood Cliffs, N.J, Prentice-Hall.

[CODD70] E. F. Codd, "A Relational Model of Data for Large Shared Databanks," Comm. ACM, Vol. 13, No. 1, June 1970, PP.377-387.

[COMP79] Computer Issue of April 1979, "Bell Labs Develops Smaller, Faster Bubble Memory." pp. 114.

[COPE73] G. P. Copeland, G. J. Lipovski and S. Y. W. Su, "The Architecture of CASSM: A Cellular System for Non-Numeric Processing," Annu. Sympos. Computer Architecture Proceeding, Dec. 1973, PP. 121-128.

[COUL72] G. F. Coulouris, J. M. Evans and R. W. Mitchell, "Towards Content-Addressing in Databases," The Computer Journal, Vol. 15, No. 2, February 1972.

[DATE77] C. J. Date, "An Introduction to Database System," Second Edition, Addison Wesley.

[DEFI73] C. Defiore and P. B. Berra, "A Data Mangement System Utilizing an Associative Memory." AFIPS Conference Proceedings, Vol. 42, June 1973, pp. 181-185.

[DEFI71] C. DeFiore, N. Stillman and P. Berra, "Associative Techniques in the Solution of Data Management Problems," Proceeding 1971 ACM National Conference.

[DEWI79] D. J. Dewitt, "DIRECT-A Multiprocessor Organization for Supporting Relational Database Management Systems," IEEE Trans. on Computers, Vol. C-28, No. 6, June 1979, pp.395-408.

[EPST80] R. Epstein and P. Hawthorn, "Design Decisions for the Intelligent Database Machine," Proceedings of NCC 49, AFPS, 1980.

[FELL67] W. Feller, "An Introduction To Probability Theory and Its Application," Vol.1, Wiley, New York, 1967.

[FLYN72] M. J. Flyn, "Some Computer Organization and their Effectiveness," IEEE Trans. on Computers, Vol. C-23, No. 2, PP. 121-132.

[GOOD81] J. R. Goodman, "An Investigation of Multiprocessor Structures and Algorithms for Database Management," Memo No. UCB/ERLM81/33 (May 1981), Electronic Research Lab., College of Engineering, University of California/Berkeley.

[GOOD80A] J. R. Goodman and A. M. Despain, "A Study of the Interconnection of Multiple Processors in a Data Base Environment," Proc. 1980, International Conference on Parallel Processing, PP. 269 - 278, Aug. 1980.

[HAWT81] P. B. Hawthorn and D. Dewitt, "Performance Analysis of Alternative Database Machine Architectures," IEEE Trans. on Software Engineering, Vol. SE.8, No. 1, Jun. 1982, pp. 61-75.

[HEAL76] L. Healy, "A Character-Oriented Context-Addressed Segment-Sequential Storage," Third Annual Symposium on Computer

Architecture PP.172 - 177.

[HEAL72] L. Healy, G.J. Lipovski and K. Doty, "The Architecture of a Context-Addressed Segment-Sequential Storage," AFIPS Conference Proceeding, Vol. 41, Part II, 1972, PP. 691 - 701.

[HSIA81] D. K. Hsiao, "Data Base Computers," Advances in Computers, Vol. 19, June 1981.

[HSIA79] D. K. Hsiao and Menon, "The Post Processing Functions of a Database Computer," OSU-CISRC-TR-79-6, OHIO State University, July 1979.

[HSIA76A] D. K. Hsiao and K. Kannan, "The Architecture of a Database Computer Part II: The Design of Structure Memory and its Related Processors," OSU-CISRC-TR-76-2, OHIO State University, OCT. 1976.

[HSIA76B] D. K. Hsiao and K. Kannan, "The Architecture of a Database Computer Part III: The Design of the Mass Memory and its Related Components," OSU-CISRC-76-3, OHIO State University, Dec. 1976.

[IBM80] IBM Corporation, "IBM 3380 Direct Access Storage Description and User's Guide," IBM Document GA26-1664-0, File No. 51370-07, 4300-07, 1980.

[KANN78] K. Kannan, "The Design of a Mass Memory for a Database Computer," Fifth Annual Symposium on Computer Architecture, April 1978, pp. 44-51.

[KIM79] W. Kim, "Relational Database Systems," Computing Survey, Vol. 11, No. 3, Sept. 1979.

[KNUT73] D. E. Knuth, "The Art of Computer Programming, Vol. 3: Searching and Sorting," Addison-Wesley, Reeding, Mass.

[LIN76] C. S. Lin, C. P. Smith and J. M. Smith, "The Design of a Rotating Associative Memory for Relational Database System," ACM Trans. Database Systems, Vol. 1, March 1976, pp. 53-65.

[LIND73] R. Linde, R. Gates and T. Peng, "Associative Processor Application to Real-Time Data Mangement," AFIPS Conference Proceeding, Vol. 42, 1973, pp. 187-195.

[LIPO78] G. J. Liposki, "Architectural Feature of CASSM: A Context Segment Sequential Memory," Fifth Annual Symp. Computer Architecture Proceedings, Palo Alto, CA, April 1978, pp. 31-38.

[MADN79] S. E. Madnick, "The Infoplex Database Computer: Concepts and Directions," Proceedings of the IEEE Compcon., Spring 1979, PP. 164 - 176.

[MADN75] S. E. Madnick, "INFOPLEX - Hierarchical Decomposition of a Large Information Management System Using a Microprocessor Complex," Proceeding of the NCC, 1975, pp. 581-586.

[MAN86] Intel Corporation, "iaPX 86, 88 User's Manual," July 1981.

[MART77] J. Martin, "Computer Database Organization," Prentice Hall, 1977.

[MENO81] M. J. Menon and D. K. Hsiao, "Design and Analysis of a Relational Join Operation for VLSI," Proceedings VLDBS, 1981, pp. 44-55.

[MOUL73] R. Moulder, "An Implementation of a Data Management System on Associative Processor," AFIPS Conference Proceeding, Vol. 42, 1973, pp. 171-179.

[OLIV79] E. Oliver, "RELACS, An Associative Computer Architecture to Support a Relational Data Model," Doctoral Dissertation, Syracuse University, 1979.

[OZKA77] A. Ozkarahan, S. A. Schuster and K. C. Sevcik, "Performance Evaluation of a Relational Associative Processor," ACM Trans. on Database Systems, Vol. 2, No. 2, June 1977, pp. 175-195.

[OZKA75] E. A. Ozkarahan, S. A. Schister and K. C. Smith, "RAP-An Associative Processor for Database Mangement," AFIPS Proceedings, Vol. 45, 1975, pp. 379-387.

[PARH72] B. Parhami, "A Highly Parallel Computing System for Information Retrieval," AFIPS Conference Proceedings, Vol. 41, Part II, 1972, pp. 681-690.

[PARK71] J. L. Parker, "A Logic per Truck Retrieval System," Proceeding IFIP Congress 1971, pp. TA4-146 to TA-4-150.

[QADA83A] G. Z. Qadah, "A Database Machine for Very Large Relational Databases," Submited for Puplication to the Parallel Processing Conference, Feb. 1983.

[QADA80] G. Z. Qadah, "A Relational Database Machine: Analysis and Design," Ph. D. Thesis Proposal, Electrical Engineering Department, the University of Michigan-Ann Arbor, 1983.

[SCHU79] S. A. Schuster, H. B. Nguyen, E. A. Ozkarahan and K. C. Smith, "RAP.2- An Associative Processor for Database and its Applications," IEEE Trans. on Computers, June 1979, Vol. C-28, No. 6.

[SHAW80] D. Shaw, "Knowledge-Based Retrieval on a Relational Database Machine," Ph.D. Thesis, Aug. 1980, Dept. of Computer Science, Stanford University.

[SLOT70] D. L. Slotnick, "Logic Per Truck Devices," Advances in Computers, Academic Press, 1970, pp. 291-296.

[STON71] H . S. Stone, "Parallel Processing with Perfect Shuffle," IEEE Transaction on Computing, Vol. C-20, No. 2, Feb. 1971, PP. 153 - 161.

[SU79] S. Y. W. Su, et al., "The Architectural Features and Implementation Techniques of the Multicell CASSM," IEEE Trans. on Computer, Vol. C-28, No. 6, June 1979.

[SU79] S. Y. W. SU, G. P. Copeland and G. J. Lipovski, "Retrieval Operations and Data Representation in a Context-Addressed Disk System," ACM SIGPLAN and SIGIR Interface Meeting, NOV. 1973, PP. 144 - 156.

[THRE78] B. Threewitt, "CDDs Bring Solid-State Benefits to Bulk Storage for Computers," June 22, 1978, Electronics.

[TSIC77] D. C. Tsichritis and F. H. Lochovsky, "Database Management Systems," Academic Press, 1977.

[TZOU80] A. Tzou, et al., "A 256K_BIT Charge-Coupled Device Memory." IBM J. Res. Development, Vol. 24, No. 3, May 1980, pp. 328-338.

[YAU77] S. S. Yau and H. S. Fung, "Associative Processor Architecture- A Survey," ACM Computing Surveys, Vol. 9, No. 1, March 1977, PP. 3 - 27.