DESIGN OF THE CADM BASED SORT/SEARCH/SET ENGINE

Vivekanand Bhat

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-87-36 September 1987

Abstract

Managing large databases involves time consuming and computationally intensive operations such as sorting, searching and various relational algebraic operations. These are traditionally performed on general purpose computers. Special hardware can be used to achieve higher speed in implementing these operations. The design of one such system for performing sort, search and set operations such as union, intersection and set difference, is described. The performance of this system is compared with the performance of general purpose computers for performing the above operations, and it is shown that our system performs considerably better. Implementation of a prototype is described.

ACKNOWLEDGEMENTS

I am very grateful to my advisor Dr. Roy Jenevein for providing invaluable guidance and support throughout the course of this work. I wish to express my gratitude to my co-supervisor Dr. Baxter Womack for his time and suggestions. I also thank Dr. Alfred Dale for his suggestions.

I wish to expresss my sincere gratitude to all the members of the Product Planning Group of Advanced Micro Devices Inc., Austin, in particular to Robert O'Dell, Robert Oliver, Brian McMinn and Zahid Ahsanullah for their suggestions and for providing some of the routines used in the simulator described in Chapter 4. My special thanks to Alice Anderson for her help in building the prototype described in Chapter 5 and for reading the first draft of this document.

My sincere thanks to Ashok Adiga and Hemendra Talesara for their suggestions and support. I also wish to thank Tarak Parikh and Prasad Sakhamuri for their help in preparing this document.

This work was supported in part by the Office of Naval Research through the grant N00014-86-k-0499.

T. Vivekanand Bhat

The University of Texas at Austin

September 1, 1987

Table of Contents

CHAPTER 1
INTRODUCTION

Diversification of computer applications is causing computer architectures to evolve in a diverse manner. Inadequacies of general purpose computers for data management had led to the development of special purpose dedicated machines. Examples of such systems are the Relational Database Machine (RDBM) [SCHW 83], the Relational Algebra Processor(RAP) [OZKA 75], Content Addressable Segment Sequential Storage (CASSM) [LIPO 75] and the DELTA [SAKA 84] .

Sorting has been identified as one of the very time consuming operations in database machines. Methods to speed up sorting in database machines have been investigated in the past, and are still being investigated. These methods use specialized hardware to achieve this goal. A discussion two of these previous approaches will follow. The solution proposed by us, is described in detail in the following chapters.

1.1 SORT PROCESSOR (SOP) OF THE RDBM

The Relational Database Machine (RDBM) [HELL 1981] from the Technical University of Braunschweig, West Germany, incorporated several hardware units to speed up time consuming and computationally intensive operations in its architecture. The overall architecture of RBDM is shown in figure 1.

The main features of this architecture are :

- Mass storage device with its own storage manager and associative data access facility.
- A multiprocessor system consisting of special function processors with common access to the main memory.
- A general purpose computer to control the different hardware components and to perform the database operations not supported by the specialized hardware.

To efficiently implement sorting operation, the *Sort Processor*(SOP) has been included in the design. This unit is designed to perform both internal external sort operations. The special purpose hardware in SOP is controlled by its own internal microprogram unit, which carries out various phases of sorting. Overall control of the sort processor is provided by a bit-slice microprocessor unit.

The SOP uses a address table list sort with merging. The algorithm has been described in the literature cited above, and the complexity has been determined to be $O(n \log_2 n)$ for a set of n records.

The tuples to be sorted are held in the common main memory of the system. The SOP will contain a table of address pointers to these tuples. This address table is built by the memory manager while the tuples are loaded from secondary to the primary memory and is transferred into the SOP using DMA transfer. During the sorting procedure, whenever the SOP needs the key in the main memory for comparison, these key bytes are read in from the main memory. At the end of the sort operation, the address pointer table in the SOP is used to reorganize the data in main memory in sorted order. This transfer of address pointers is done using handshake mechanism. A bit in the address pointer table entry indicates if the entry is a duplicate of its predecessor. If duplicate elimination is required, this bit can be used to ignore the corresponding tuples.

The main hardware components can be described as follows :

- The address table contains the tuple address, link address between tuple during the sort and the duplicate bit.

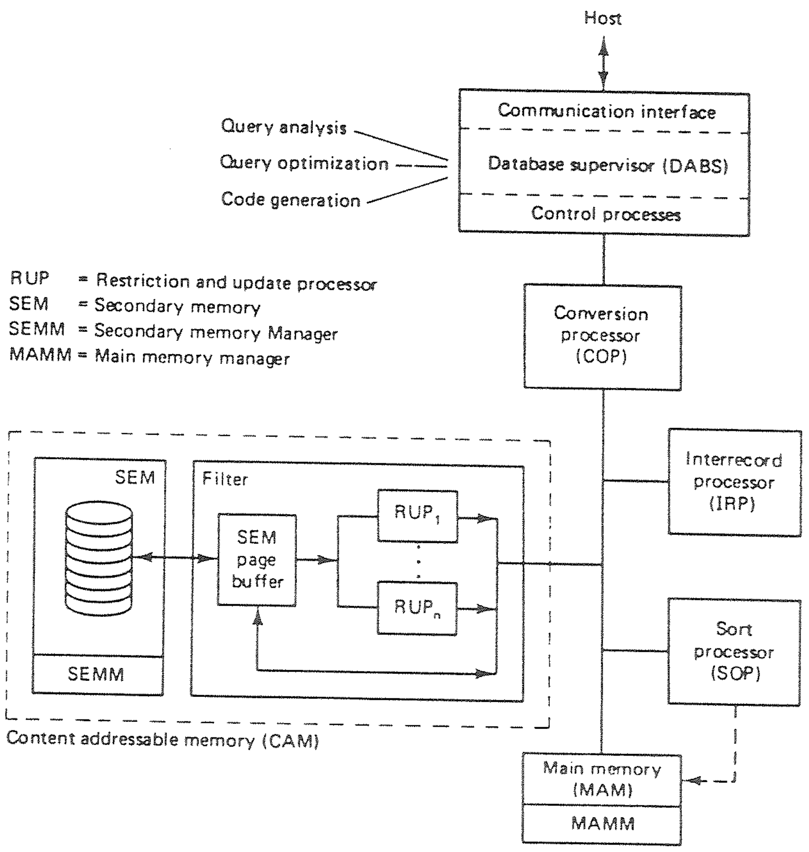

Host
Communication interface
Query analysis
Query optimization
Code generation
Database supervisor (DABS)
Control processes
RUP = Restriction and update processor
SEM = Secondary memory
SEMM = Secondary memory Manager
MAMM = Main memory manager
Conversion processor (COP)
SEM
Filter
SEM page buffer
RUP_1
RUP_n
SEMM
Interrecord processor (IRP)
Sort processor (SOP)
Content addressable memory (CAM)
Main memory (MAM)
MAMM

Figure 1. Architecture of the RDBM

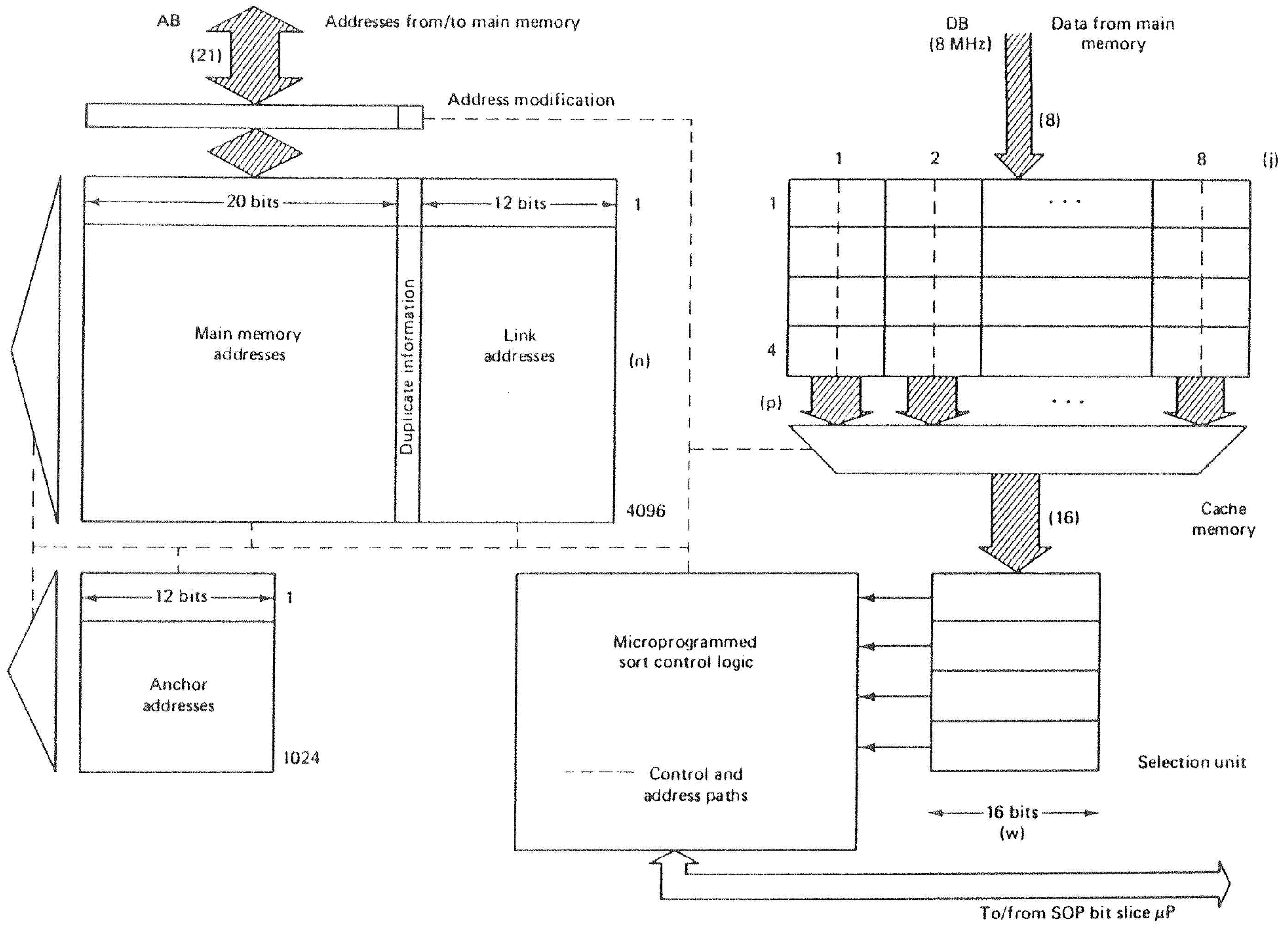


Figure 2. Block diagram of the SOP

- A cache memory of 4 X 16 bytes is used to store the current portion of the keys being compared.
- A merge selection unit implemented in schottky TTL for speed.
- Microprogram store, of 256 words of 128 bits each.

External sorting procedure uses the SOP to sort portions and keeps them in main memory. These sorted subsections are then merged using the merge unit in the SOP.

1.2 RELATIONAL ENGINE OF THE DATABASE MACHINE DELTA

The Relational database machine Delta [SAKA 84, ITOH 87] is under development at the Institute for New Generation computer Technology, Japan. The overall architecture of this machine is shown in figure 4.

The machine has an interface unit (IU), a control unit (CU), schedule unit (SU), hierarchical memory control unit (HMCU) and multiple relational engines (RE).

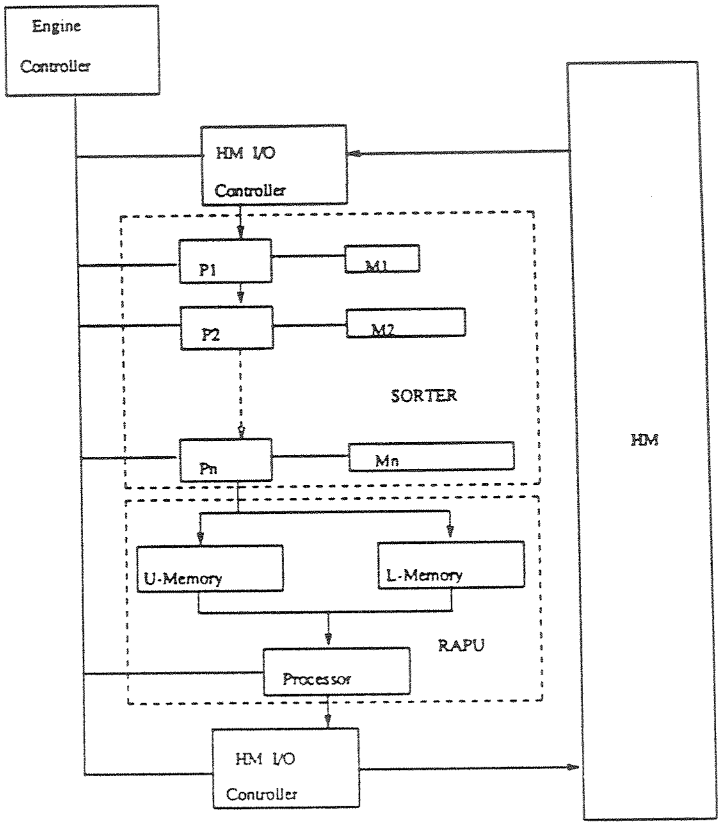


Figure 3. RE Hardware configuration

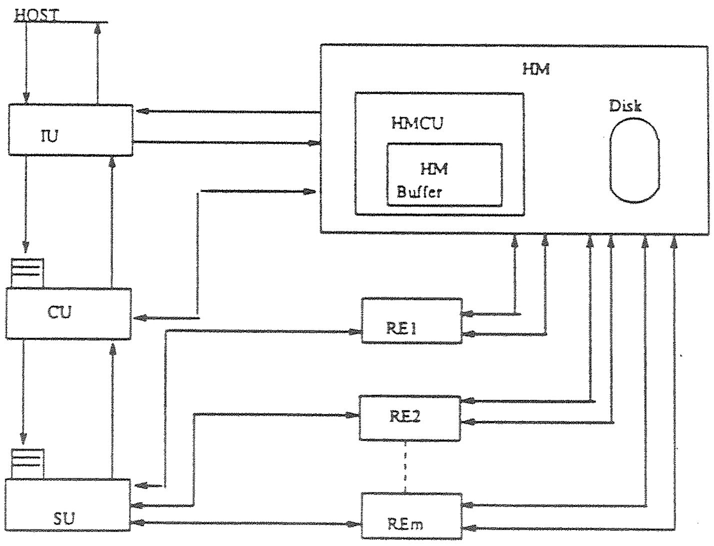


Figure 4. Block diagram of the database machine Delta

The IU, CU, and HMCU are single CPUs. The HMCU includes a disk drive (HM Disk) for mass storage of relations. The IU receives a command from the host, and sends it to the CU queue. When it receives a signal that a particular operation has been completed, the IU directs the output to the host. The CU takes the commands from its queue in FIFO order and interprets them and converts them into internal commands. It asks the HMCU to obtain relations from disk to buffer memory and sends commands to the SU internal queue. When the processing is completed, it sends signals to IU. The SU takes commands from its queue and divides the processing among all the inactive engines (REs). The REs take commands from SU and execute them. The results are sent to the HM.

The configuration of REs is shown in figure 3. It is composed of a sorter and a Relational Algebra Processing Unit (RAPU). The sorter sorts the data and the RAPU performs relational operations on the stream of data from the sorter. The data is processed very fast by the RAPU on the fly. The data transfer speed is 3 MBytes/sec. A pipelined 2 way merge sort algorithm [TODD 78] is used to achieve sorting in O(n) for n records.

The sorter has 12 levels of processors, each with dedicated memory space. The memory size at each processor increases at each level as follows :

Memory size for processor 1 : 32 bytes

Memory size for processor i : 2(size of memory for processor i-1)

Memory size for processor 12 : 64 Kbytes

The sorter is thus capable of sorting 64KBytes of data at a time. Maximum length of each record is 16 bytes.

The RAPU consists of two memories - U-memory and L-memory. These are used to store sorted data and a comparator is used to select the output data satisfying the given condition. Each of these memories is 64KBytes in size. The REs also have an I/O controller for relation storage in HM, and an engine controller for overall control.

1.3 A PARALLEL RELATIONAL DATABASE SYSTEM :

With the improvement in technology, computer systems with higher computational power are being developed. But the external memory has continued to be a major limiting factor in these systems, mainly because of the inability to develop faster external memory systems. This factor has been a bottleneck in database systems of today, which are required to handle increasingly large volumes of data. With the objective of developing an external memory system commensurate with the computational power of next generation host machines, an architecture was proposed in [BROW 85]. This architecture lends itself to parallel access to databases and to parallel operations on data objects that are being streamed from the secondary storage toward the host.

Sorting and searching, along with set operations form a very significant part of the computations performed in a database system. These operations are also extremely time consuming and computationally expensive. These observations have lead to the inclusion of a functional unit for efficiently performing these operations, in the above system. This unit is based on the Content Addressable Data Manager (CADM) [AMD 86] chip from Advanced Micro Devices. Design of this Sort engine and the algorithms to perform its functions are described in detail in the following chapters.

1.4 ARCHITECTURE OF THE SYSTEM :

The architecture of this system is as shown in figure 5. The architecture is partitioned into four major levels.

- Host processors, which can be general purpose or special purpose processors.

- A set of "Node Mappers" which make an associative translation from the requested objects (eg. relation names, attribute name-value pairs and tuples) to base (I/O) nodes where the objects are stored, by generating the appropriate routing tag for the Interconnection Network.

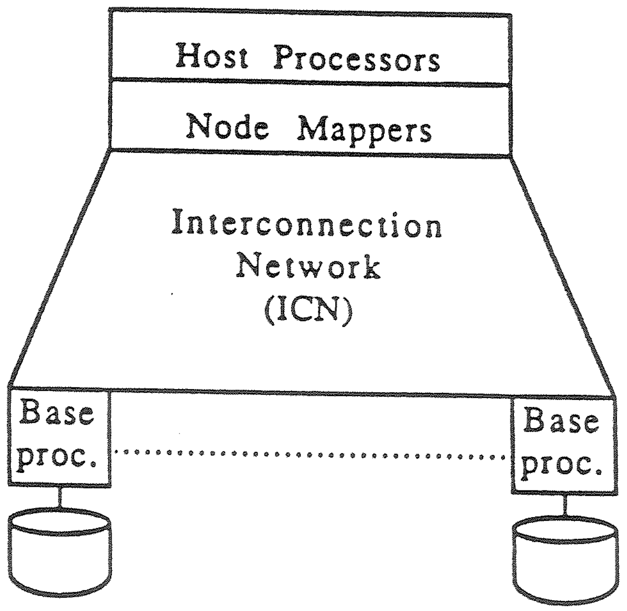


Figure 5. Architecture of the Parallel I/O System

- An Interconnection Network which couples host and base(I/O) processors. The interconnection topology is based on the KYKLOS [MENE 85] multiple tree topology. The switch nodes of this network also incorporate logic and buffering to support merge operation on data streams. The suitability of using this interconnection network for performing join operations has been investigated in [JENE 87].

- I/O nodes each consisting of a general purpose computer, associative disk cache, and the **Sort Engine** along with moving head disks for storing data. The general purpose computer will be a VAX 8250.

1.5 I/O NODE ARCHITECTURE :

The architecture of the I/O node is shown in figure 6. The VAX 8250 is used to control the I/O node system. It has 4 MBytes of local memory. Various functional units are connected together on the high speed BI bus, which is the backplane bus of VAX 8250. This bus operates at 13.3 MBytes/sec transfer rate. The Network Interface Unit interfaces to the Interconnection Network.

The Disk System consists of a track based Disk Cache, a data filter, mass storage with disk controller and a microprocessor to control the operations of the Cache subsystem.

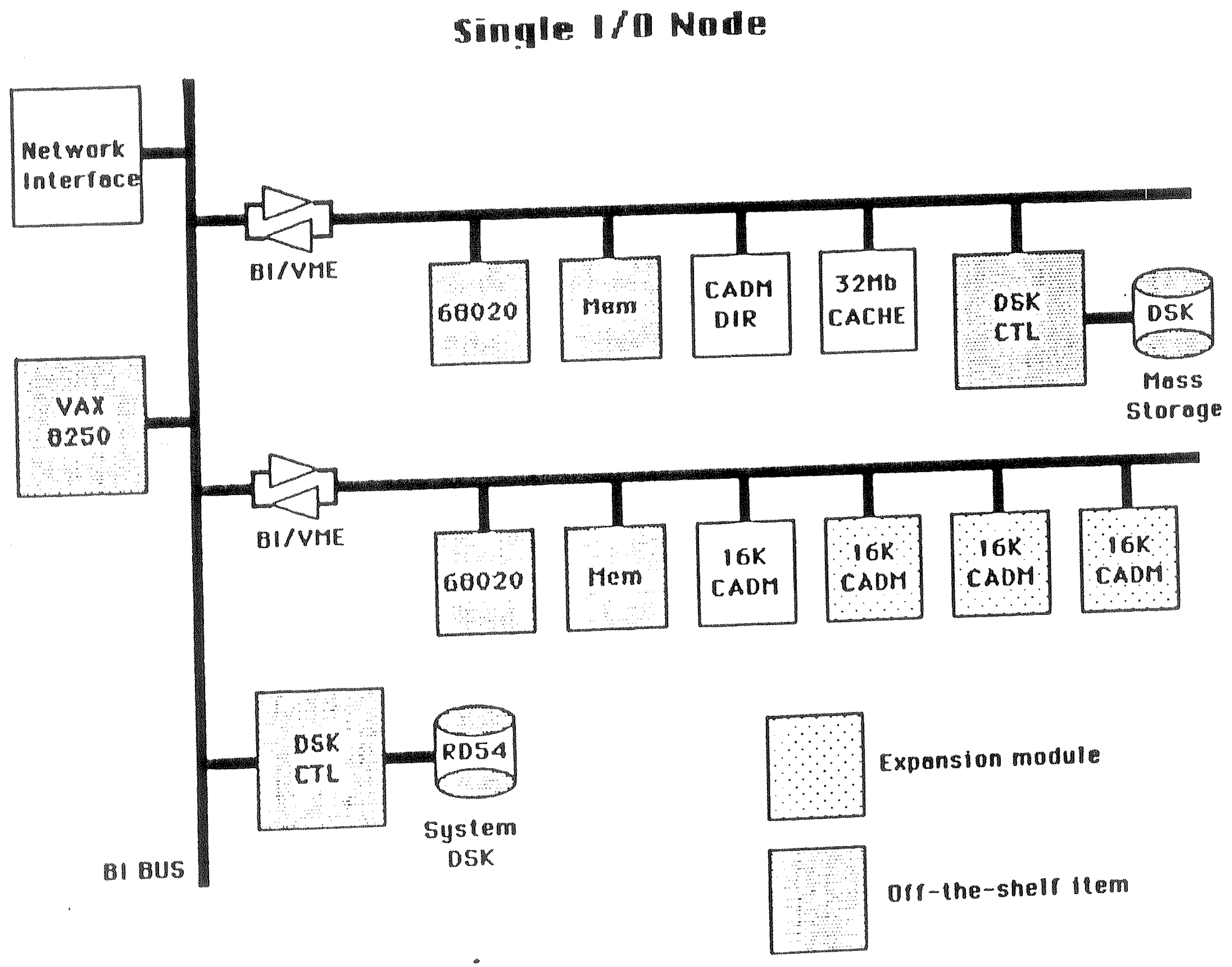


Figure 6. Architecture of the I/O node

Mass storage consists of between one and four 500 MByte disk drives. The disk cache has capacity of 32 MBytes. The cache is track-based, and so the minimum unit for disk transfers is a track. The data filter filters the data from the disk on the fly. MC68020 processor controls the operation of this subsystem.

The **Sort Engine** is the heart of the I/O node design. It performs Sorting, Searching, Selection and set operations such as Intersection, Union and Set Difference. This engine is based on the CADM chips. The engine is controlled by a MC68020 microprocessor and also has RAM and ROM memory.

The local bus in these subsystems is the VME bus. There are BI/VME converters that connect these two busses together. This bus structure allows the various subsystems to function concurrently and provides substantial speedup due to overlap of their respective operations.

Chapter 2 gives a brief description of the CADM. Chapter 3 describes the architecture of the Sort engine and the various algorithms for performing the different functions. A simulator was developed to evaluate the performance of this engine. The structure of this simulator is described in Chapter 4. The results of the simulation are also described. Chapter 5 describes a test set-up that was implemented, and also the detailed description of hardware for the proposed system. Some conclusions drawn from this study are in Chapter 6.

CHAPTER 2

CONTENT ADDRESSABLE DATA MANAGER

Mapping certain time consuming operations that are usually performed in software onto fast hardware has produced efficient solutions to these problems. Advances in the field of VLSI has increased this trend. One such device is the *Content Addressable Data Manager* (CADM - Am95C85) [AMD 86] designed by Advanced Micro Devices (AMD). This support chip is capable of sorting, searching and stack operations. It combines the advantages of Content Addressable Memory with the flexibility of Random Access Memory. It contains a microprocessor that can work independent of the host. Some prominent features of this chip are :

- On-chip microprocessor that controls host independent processing and manipulation.
- 1 kbyte RAM on chip. Up to 256 chips can be cascaded together.
- Software programmable record width.
- Stack mode of operation allows insertion and deletion of data at any location in the CADM.
- Powerful and versatile instruction set.

- Clock frequency range of 1 to 16 MHz.

2.1 FUNCTIONAL DESCRIPTION :

The programmer's view and the block diagram of the CADM are shown in figures 7 and 8 respectively. The address space of the CADM consists of a mask area, a record area and an input buffer area. A record consists of 1 to 255 bytes of key and 0 to 255 bytes of pointer. Any bit of the key can be masked, by setting the corresponding bit in the mask field. The mask field is of the same size as the key field. The input buffer space is used to store a record prior to any operation.

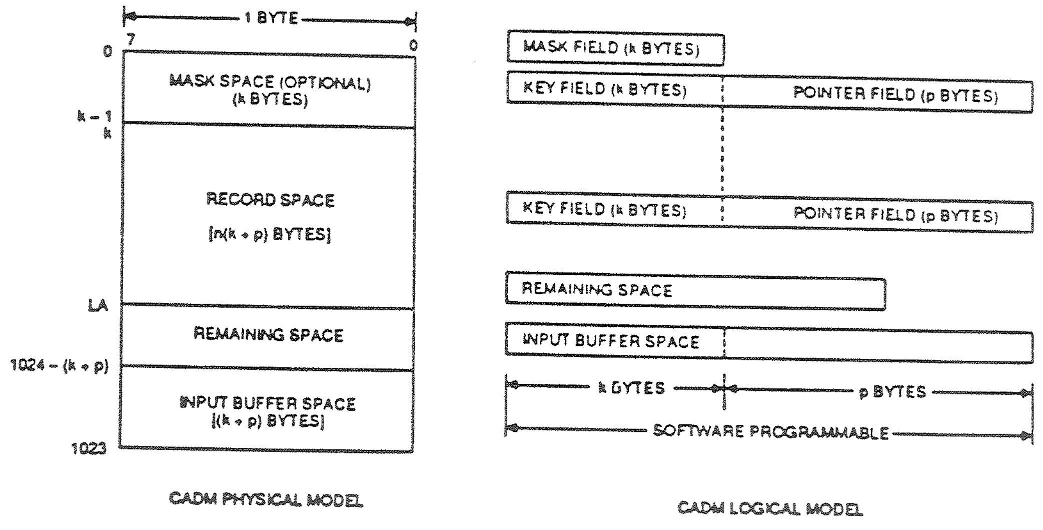


Figure 7. Programmer's view of the CADM

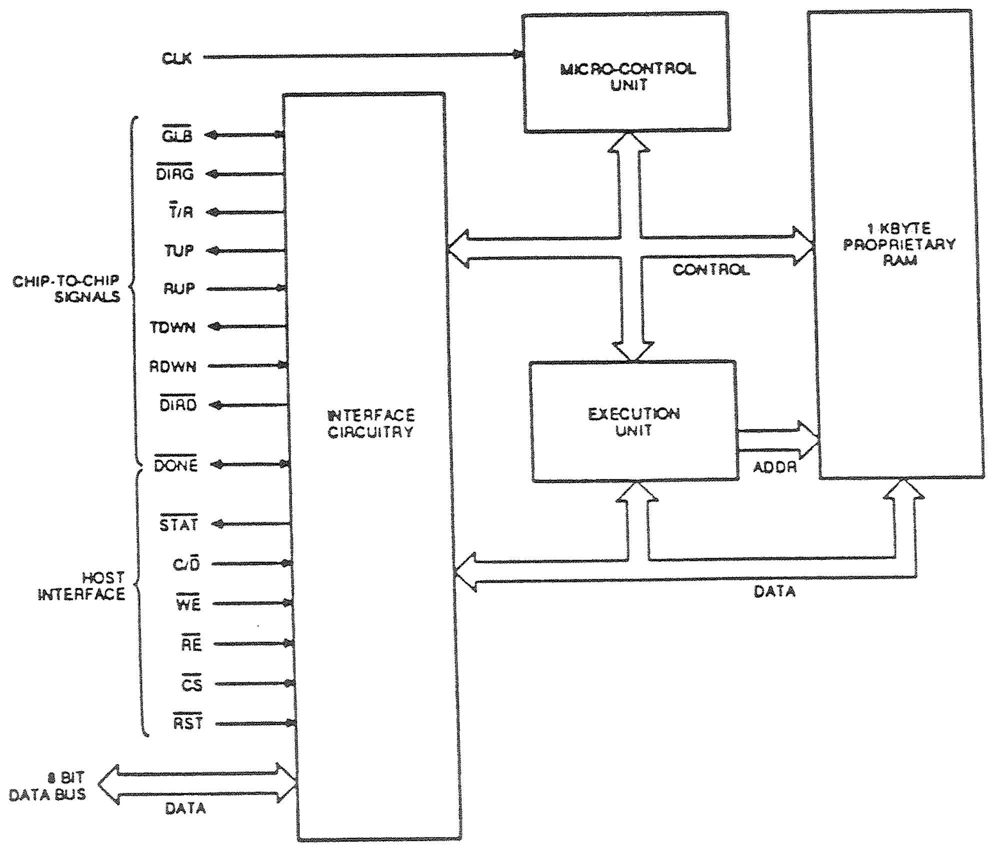


Figure 8. Block diagram of the CADM

2.2 ADDRESSING MODES :

There are three addressing modes in CADM. The *Auto Increment* mode allows rapid reading or writing of data, by automatically incrementing the internal address pointer after every read or write. The *Stack Mode* allows insertion or deletion of data anywhere in the CADM. A read in stack mode is equivalent to a pop operation and a write is equivalent to a push. After a push or pop from a particular location, CADM physically moves all data below that location. The *CAM Mode* allows searching operations for a particular record, provided the data is sorted. When CADM chips are cascaded, the searching takes place simultaneously in all chips. Each CADM maintains all the pointers required to support these modes. The *Address Pointer*, from which data can be accessed, is write only.

2.3 CADM ARCHITECTURE :

The block diagram of the CADM is shown in figure 8. The CADM communicates with the host via an eight bit data bus and 7 control lines (host interface signals). It has the 1 KByte RAM, along with a micro control and execution units. The chip to chip signals, allow it to communicate with other chips which are cascaded with it. The host interface signals consist of the reset(RST), read enable (RE) and write enable (WE), chip select (CS) and command/data (C/D). The maximum transfer rate for data into and out of the CADM is 2.6 MBytes / sec.

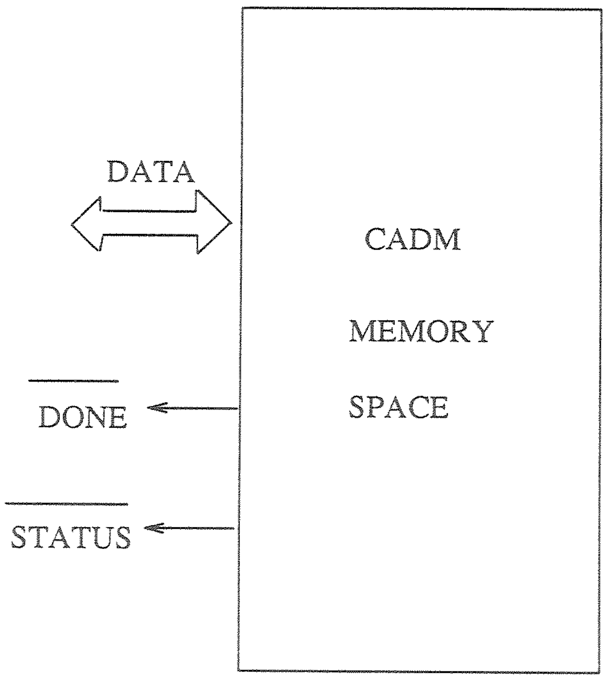


Figure 9. CADM Model for Simulation

There are two other very important signals that are essential in interfacing the CADM to the host. The DONE signal indicates the end of a particular operation in CADM. The next operation can start only after this signal becomes true.

The STATUS signal indicates an exception condition resulting from a particular operation in the CADM. Particularly, during a search operation, STATUS becomes true if the key being searched for, does not exist in the CADM. In this case, the address pointer will point to the record with the next higher key value, present in CADM. Also, the DONE pin never becomes true if an attempt is made to write data into CADM before specifying the key and pointer lengths or if as a result of any operation, access is made to a data location beyond the valid data limit. This applies to the search also.

Up to 256 CADMs can be cascaded to form a single bank. Due to capacitance effects, the maximum clock frequency has to be reduced to 12 MHz if more than 16 CADMs are to be cascaded. Two levels of buffering are required in this case. The data bus and the control signals of each block of 16 CADMs have to be connected to the data bus and signals from the other banks through buffers. The signals for the entire set of CADMs have to be connected to the host through another set of buffers.

For the simulation of the sort engine and for the design and analysis of the various algorithms used to perform the different functions, the model shown in figure 9 is used.

2.4 CADM PERFORMANCE :

CADM can perform two types of sort operations. The *On-Line sort* (SON) inserts the records one at a time, in its place such that the entire list is in sorted order. In *Off-Line sort* (SOF), data is first inserted and can be sorted without the host having to wait for the operation to end. Off-line sort performance of CADM is data dependent. The best case performance is obtained when the data is previously sorted, with keys that do not have matching most significant byte. The worst case is that of reverse sorted data, with keys that differ only in their least significant byte. The equations for time required for both cases is :

$$T_{SB} = \frac{9 + N\left[20 + 6(K+P) + 8.5\left(\left\lfloor \log_2 (n+1) \right\rfloor\right)\right]}{F}$$

$$T_{SW} = \frac{9 + N\left[21 + \left(9 + \left\lceil \frac{N}{n} \right\rceil\right)(K+P) + \left(\left\lfloor \log_2 n \right\rfloor + 1\right)(5.5 + 3.3K)\right]}{F}$$

Where :

N = Total # records. T_{SB} = Time for sort(best case)

n = # records in each chip. T_{SW} = Time for sort(worst case)

K = # bytes per key. P = # bytes per pointer.

F = Frequency of the CADM clock.

The best case sort is O(N) and the worst case is O(N^2), where N is the number of records at the input. CADM performs binary search on the sorted data. Search is executed in parallel in all the chips that are cascaded. Hence the search is O(log n), where n is the number of records in each chip. The equation for the time required for a find (search) operation is :

$$T_F = \frac{3 + 5K + 8.5\left(\left\lfloor \log_2 n \right\rfloor + 1\right)}{F}$$

The main reasons for designing the Sort Engine with the CADM was its superior performance in terms of speed and the ability to manipulate data, independent of the host. Simulations performed at AMD and by us have demonstrated its performance. When operating at 16 MHz, for sorting data, it was at least 25 times faster than a VAX 11/780 and about 1.5 times faster than a CRAY X/MP. For searching, it is essentially faster than conventional uniprocessor machines, and the search time remains practically same as the number of chips that are cascaded increases, because searching is done in parallel in all chips at the same time.

CHAPTER 3
SORT ENGINE

3.1 ARCHITECTURE OF OUR SORT ENGINE :

The block diagram of the Sort Engine is shown in figure 10. The MC68020 microprocessor controls the sort engine. The RAM memory is primarily used for performing operations on large volumes of data, when the size of the data is greater than the CADM space available. The local bus for the engine is the VME bus [MOTO 87]. This bus was chosen because of its high data transfer rate and the easy availability of modules for building the system. The ROM houses the programs for performing various functions of the engine.

This engine performs the following functions :

- Sorting data up to 20 times the size of the CADM memory available.
- Searching for a particular record by its key field.
- Selection (equal(==), not equal(!=), less than(<), less than or equal(<=), greater than(>), greater than or equal(>=) and selecting a range of data).
- Intersection of two lists, whose combined size is 20 times that of the CADM memory available.

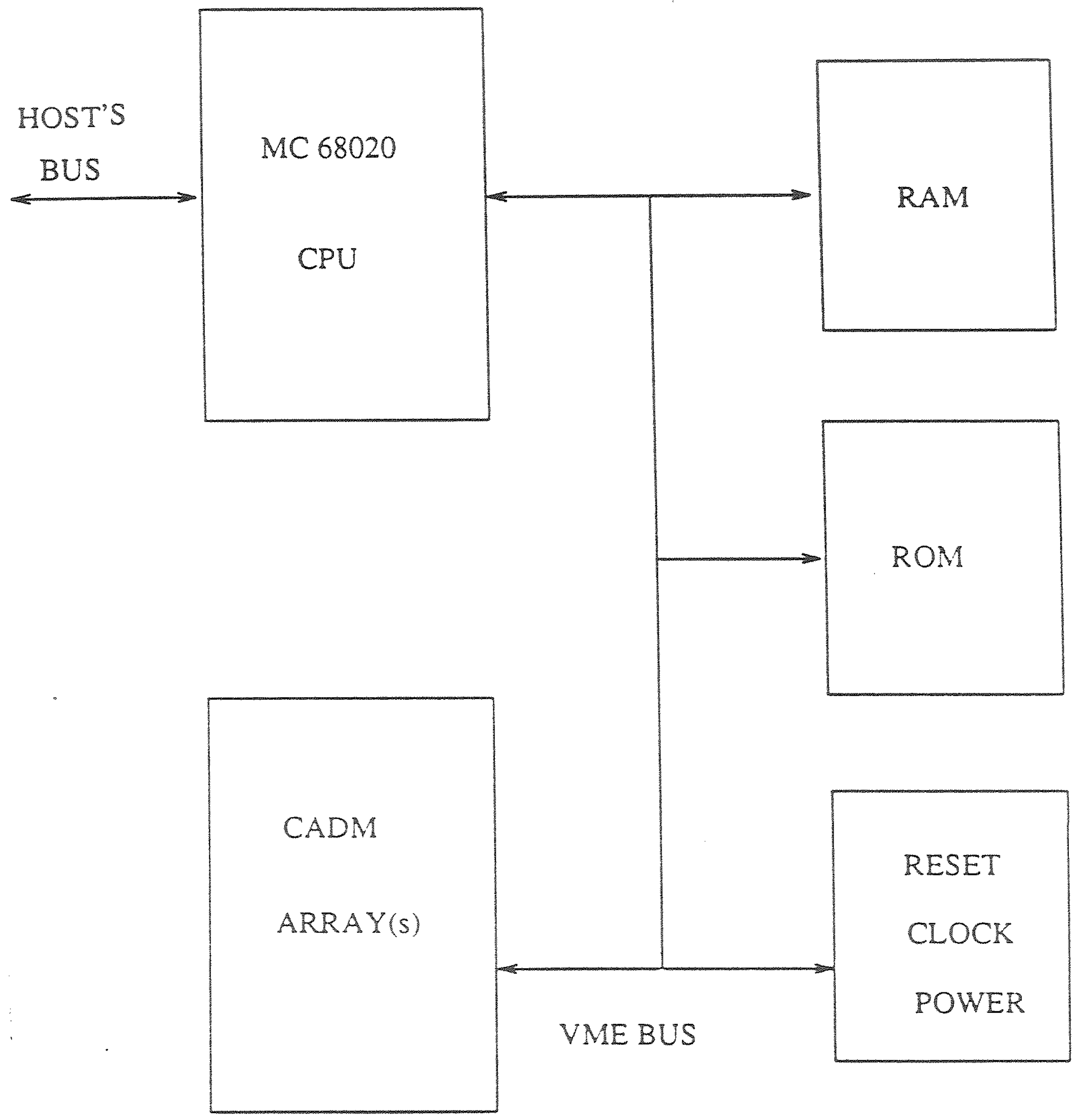


Figure 10. Block diagram of the Sort Engine

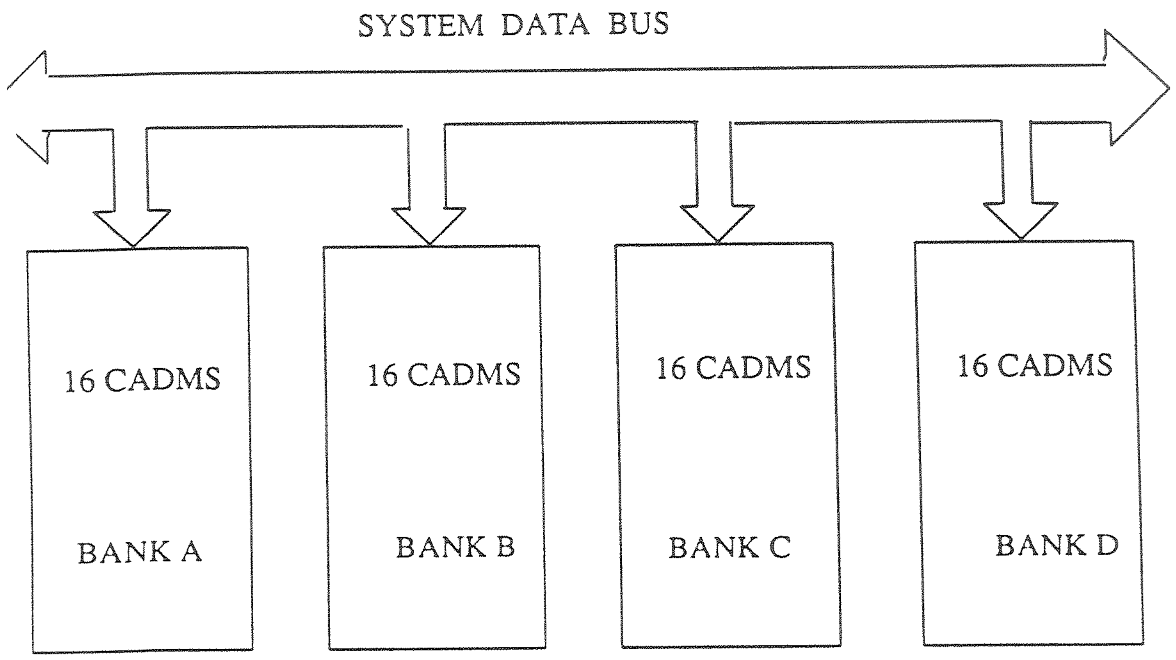


Figure 11a. CADM configuration A

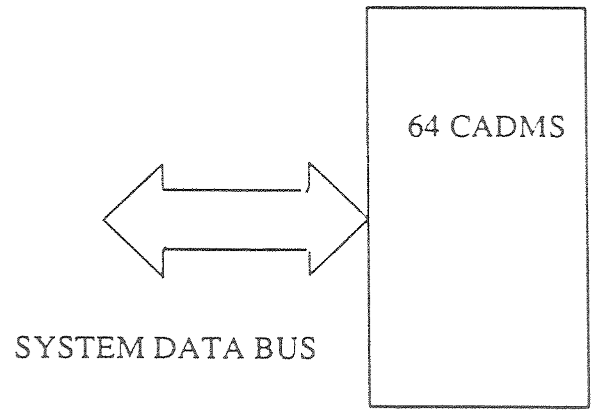


Figure 11b. CADM configuration B

- Union of two lists, whose combined size is 20 times that of the CADM memory available.
- Difference of two lists, whose combined size is 20 times that of the CADM memory available.

Speed of operation and available memory address space dictated the choice of the processor to control the operation of the sort engine. Microprocessors like the MC6809, MC68000 and MC68020 were considered and the MC68020 was chosen because of its higher computing power. The following table compares the clock speeds and the amount of memory that these microprocessors can address.

Processor	**Clock**	**Memory**
6809	2MHz	64KBytes
8051	12MHz	64KBytes
68000	16MHz	2^{25}Bytes
68020	25MHz	2^{34}Bytes

Table 1. Clock speeds and memory capacities of various microprocessors.

Due to the capacitance effects, the maximim frequency of operation of the CADM has to be reduced from 16 MHz to 12 MHz when more than 16 CADMs are cascaded together. Due to this reason, two design alternatives were considered for the CADM bank, as shown in figures 11a and 11b. The arrangement in figure 11b has a single bank of 64 CADMs operating at 12 MHz. The arrangement in figure 11a has 4 independent CADM banks of 16 CADMs each. When the arrangement shown in figure 11a is used, sorting of data can be done simultaneously after each bank is loaded, since they can operate independently. This arrangement is attractive due to the quadratic order of complexity of the CADM sort.

3.2 ALGORITHMS FOR SORT ENGINE OPERATIONS :

Two sets of algorithms need to be considered here. The first set is used when the size of input data is less than or equal to the CADM memory size available. These algorithms will be referred to as *Algorithms for small data volumes*. If the CADM bank configuration of figure 11a is used, the size of data for these algorithms must be less than or equal to 16K bytes. The second set of algorithms operate on large volumes of data, i.e., when the size of input data is greater than the CADM memory size available. These algorithms will be referred to as *Algorithms for large data volumes*. These algorithms divide the input data into segments of size equal to the CADM size, and then use the CADM and also the MC68020 to perform the various functions listed earlier. The aim of these algorithms is to utilize the CADM where appropriate.

Two basic operations in CADM are sorting and searching. As indicated earlier, sorting can be *On-Line* (SON) or *Off-Line* (SOF). The flow chart for SOF, Search and for initializing the CADM are given in Figures 12a, 12b, and 12c. Details of CADM commands used here are given in [AMD 86]. The initialization sequence shown consists of Reset, Loading the key and pointer length and the last valid address in CADM. Mask bytes can also be set if needed. Following reset command, a read command can be issued to obtain the number of CADMs in the array.

For Off-Line sort, the records are loaded after the initialization sequence using the *Load Unsorted Data* (LUD) command. Then, the SOF command is issued to sort the data. The DONE signal becomes true after completion of sorting. For On-Line sort, the SON command is issued, followed by one or more records. The next record can be inserted after the DONE signal becomes true.

For search, the *find* command is issued, followed by the key to be searched for and data in the CADM must already be sorted. After DONE becomes true, the address pointer points to the first byte of the key field of the record found, if STATUS is false. Otherwise, the address pointer points to the record with the next higher key value. Series of find commands can now be issued to get all records with the same key. To find if the address pointer is pointing to the last record in CADM after a search, a *next* (NXT) command can be issued. The STATUS becomes true, if the end of valid data has been reached.

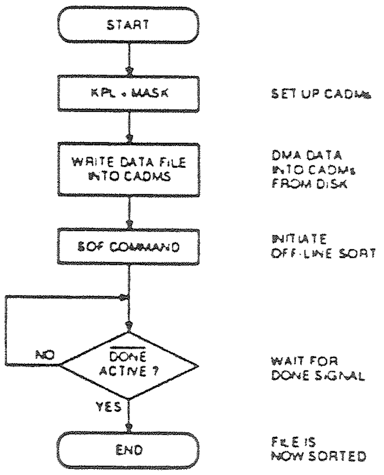


Figure 12a. SOF flow chart

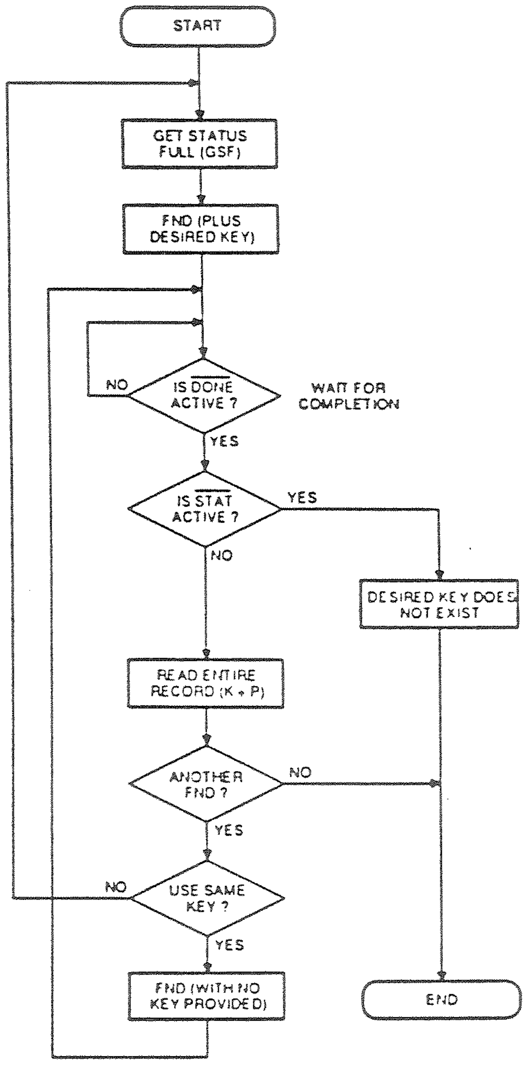

START
GET STATUS FULL (GSF)
FND (PLUS DESIRED KEY)
NO
IS $\overline{\text{DONE}}$ ACTIVE ?
WAIT FOR COMPLETION
YES
IS $\overline{\text{STAT}}$ ACTIVE ?
YES
NO
DESIRED KEY DOES NOT EXIST
READ ENTIRE RECORD (K + P)
ANOTHER FND ?
NO
YES
NO
USE SAME KEY ?
YES
FND (WITH NO KEY PROVIDED)
END

Figure 12b. Search flow chart

3.3 ALGORITHMS FOR SMALL DATA VOLUMES :

Sorting and Searching are trivial in this case, as they are commands available in the CADM. Algorithms for performing Selection, Intersection, Union and Difference are given in the following sections. For binary operations, both input lists will be in CADM banks for configuration in figure 11b. For the other configuration, one of the lists will reside in main memory after being sorted. Result of the operations will either be in the CADM array or in the output buffer in the main memory of the microprocessor.

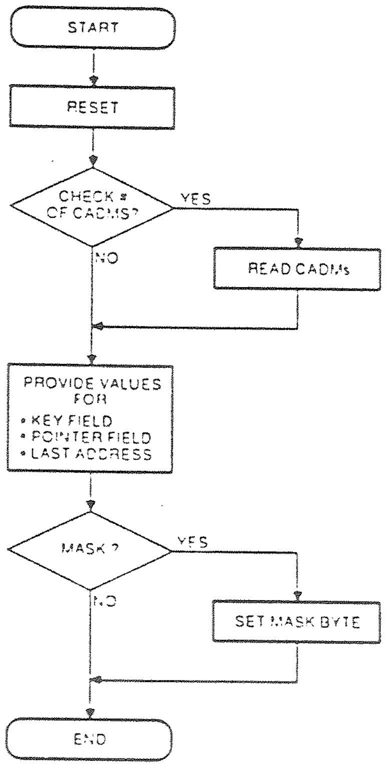


Figure 12c. CADM initialization

3.3.1 Union :

Let the lists whose union is to be found, be A and B. Let list B be the smaller of the two lists.

```
Step1 : Sort lists A and B in the CADM off-line.

Step2 : numrecs = number of records in list B.

Step3 : i = 0.

        DONE = FALSE.

        While i < numrecs && DONE = FALSE  DO

                Get Record$_i$ from list B.

                Perform Search for Record$_i$ in CADM (list A).

                if Record$_i$  not present in CADM (list A).

                        PUSH Record$_i$ into CADM (list A).

                If END of CADM (List A),  DONE = TRUE.

                i = i+1

        End DO

Step4 :         Append rest of List B to CADM (List A).
```

It should be noted here that if configuration of figure 11b is used, the input lists have to be loaded into CADM and sorted sequentially. After sorting, one of the sorted lists (the smaller of the two) must be brought back into the RAM memory of the sort engine, after the first step. If the configuration of figure 11b is used, the two lists can be sorted simultaneously in two CADM banks, and the sorted list need not be brought back into RAM memory as in the other case. This applies to all the binary operations performed in the Sort engine. CADM array A contains Union of lists A & B. This requires that CADM bank A be big enough to hold both lists A and B (worst case).

3.3.2 Intersection :

Let the lists whose intersection is to be found, be A and B.

```
Step1 : Sort lists A and B in CADM off-line.
Step2 : numrecs = number of records in list B.
Step3 : i = 0.
        DONE = FALSE.
        While i < numrecs && DONE = FALSE  DO
                Get $Record_i$ from list B.
                Perform *Search* for $Record_i$ in CADM (List A).
                If $Record_i$  present in CADM (List A)
                        Put $Record_i$ into Output Buffer.
```

```
                  If END of CADM (List A),  DONE = TRUE.
                  $i = i+1$
            End DO
```

The output buffer is in RAM, which now contains the intersection of A and B.

3.3.3 Set Difference :

Let the lists whose difference is to be found be A and B.

```
Step1 : Sort lists A and B in the CADM off-line.
Step2 : numrecs = number of records in list B.
Step3 : $i = 0$.
        DONE = FALSE.
        While $i <$ numrecs && DONE = FALSE  DO
              Get $Record_i$ from list B.
              Perform *Search* for $Record_i$ in CADM (List A).
              if $Record_i$ present in CADM (List A)
                    POP $Record_i$ from CADM (list A).
              If END of CADM (List A),  DONE = TRUE.
        i = i+1
    End DO
```

Now, CADM array A contains (A-B).

The three Algorithms described above, assume that the input lists do not have duplicates in them. This assumption assures that the output does not contain duplicates.

3.3.4 Selection :

The list of records is loaded into the CADM. The list is sorted off-line. Depending on the selection criterion, one of the following operations is performed :

- **= Key** : If the criterion is to select all records with the given key, a *Find* command is issued with the key on which the selection is required. The record found, is read into the output buffer. The *Find* command is repeatedly issued with the same key, until the CADM returns with a Not-Found signal. The record found each time is read into the output buffer. At the end, the output buffer will contain the required result.

- **!= Key** : If the criterion is to select all records other than the ones with the given key, a *Find* command is issued with the key on which the selection is required. The record found, is popped out of the CADM. The *Find* command is repeatedly issued with the same key, until the CADM returns with a Not-Found signal, and the records found each time are popped out. At the end, CADM will contain the required result.

- **>= Key** : If the selection criterion is to select all records with keys greater than or equal to the given key, a *Find* command is issued with the key on which the selection is required. Starting from the position where the address pointer of the CADM points after the search, all the records are read into the output buffer, until the end of CADM. Now, the output buffer will contain the required result.

- **> Key** : If the selection criterion is to select all records with keys greater than the given key, a *Find* command is issued with the key on which the selection is required. The *Find* command is repeatedly issued with the same key, until the CADM returns with a Not-Found signal. Starting from the position where the address pointer of the CADM points after this operation, all the records are read into the output buffer, until the end of CADM. Now, the output buffer will contain the required result.

- **<= Key** : If the selection criterion is to select all records with keys less than or equal to the given key, a *Find* command is issued with the key on which the selection is required. The *Find* command is repeatedly issued with the same key, until the CADM returns with a Not-Found signal. Starting from the position where the address pointer of the CADM points after this operation, all the records up to the end of CADM are popped out of the CADM. Now, the CADM contains the required result.

- **<Key** : If the selection criterion is to select all records with keys less than the given key, a *Find* command is issued with the key on which the selection is required. Starting from the position where the address pointer of the CADM points after this operation, all the records up to the end of CADM are popped out of the CADM. Now, the CADM contains the required result.

- **>Key$_1$, <Key$_2$** : If the selection criterion is to select all records with keys greater than key$_1$ and less than key$_2$, a *Find* command is issued with the key$_2$. Starting from the position where the address pointer of the CADM points after this operation, all the records up to the end of CADM are popped out of the CADM. Now, a *Find* command is issued with the key$_1$. All the records from the address pointer of CADM after this operation till the end of CADM are read into the answer buffer, which is the required answer.

The last three operations involve excessive reorganization of data in CADM using POPs and PUSHes. This could have been avoided if the Address pointer of the CADM could be read (it is write only).

3.4 ALGORITHMS FOR LARGE DATA VOLUMES :

Here, the size of the input data is greater than the available CADM space. These algorithms divide the input data into segments of size equal to the CADM memory size and use the control processor and also the CADM to perform the required operation on the data.

3.4.1 Algorithms for Sorting :

Three algorithms are considered here. All of these algorithms use the CADM to sort smaller portions of the input data. They differ in the merge phase where the sorted subsegments are merged.

3.4.11 Algorithm A:

This algorithm uses the CADM repeatedly to sort portions of the input file in the first phase. Each of these portions whose size is less than or equal to the CADM buffer size, will be put in blocks of memory called bins. In the second phase, which is the merge phase, two way merge [KNUT 73] is repeatedly performed on adjacent sorted bins, to get the final result. The merge phase uses the control processor and the RAM memory.

Phase I (Sort Phase):

Step1 : Number of bins = Input size / CADM size.

Step2 : Load each bin into CADM and sort off line. Load sorted bins back into their place.

Phase II (Merge Phase):

Step i (i= 1,2,....(Number of bins -1)): Merge adjacent pairs of bins.

3.4.12 Algorithm B:

This algorithm [AMD2 86] uses the CADM in the sort phase as well as the merge phase. In the sort phase, each record in the input is copied to another block of memory to append a bin tag at the end of each record. The bin tag is a number that indicates the bin number to which the record belongs. Then each bin is sorted using the CADM. During the merge phase, these bin tags are used to obtain the final list in sorted order. Let c be the # records in each CADM chip and j be the number of chips in the array (16 for CADM configuration-A and 64 for configuration-B).

Phase I (Sort Phase):

Step1 : b = Number of bins = Input size / CADM size.

Initialize counter[i] = 0 where i = 0 to (b-1).

Step2 : Copy input reccrds one at a time and append bin tag.

Step3 : Load each bin into CADM and sort off line. Load sorted bins back into their place.

Step4 : Load (c*j)/b records into CADM and sort off line.

Phase II (Merge Phase):

Step1 : Pop out (c*j)/b records from CADM into the output buffer. For each record popped, Increment counter[i] where i is its bin tag.

Step2 : From each bin, insert s records into CADM using SON, where s is the value of the counter for that bin. Reset counter to 0. If there are no records to insert, go to Step3. Otherwise go to Step1.

Step3 : Pop out remaining records from CADM to the output buffer and terminate the algorithm.

3.4.13 Algorithm C:

This algorithm is a variation of algorithm B. In Step2 of the sort phase of algorithm B, we have to copy all the bytes in the input for appending bin tag. This will take considerable time. If the records in the input have an unused "hole" at the end of each record, we can write the bin tag in this "hole" directly instead of copying all the records one at a time. This results in a significant saving in time. Rest of the algorithm remains same.

It should be noted that during the first (sort) phase, the operation differs slightly depending on the CADM array configuration used. If the configuration of figure 11a is used, four bins of 16K bytes each can be sorted simultaneously, in four different CADM banks. For the other configuration, 64K bytes of data are sorted at once.

3.4.2 Algorithm for Selection :

The input data is sorted using the algorithms described before. Depending upon the type of selection operation, binary search is performed on the sorted data in RAM, using the control processor. Another method would be to use the *Find* command in CADM, by loading each sorted bin into it. The transfer rate makes this method more time consuming than the other methods. Depending on the selection criterion, one or two binary search operations are needed.

3.4.3 Algorithms for Set operations :

Two different strategies can be used to perform these operations. In the first method, one of the lists (for example list A) is sorted using the CADM. Then the following operations are performed :

Method I :

```
Step1 : b = # bins in list A. (Size of list A / CADM Size)

Step2 : numrecs = number of records in list B.

Step3 : For i = 0 to numrecs DO
            Get $Record_i$ from list B.
            For j = 0 to b DO
                Load $Bin_j$ of list A into CADM.
                Perform Search for $Record_i$ in CADM.
                Perform required operations.
            End FOR
        End FOR
```

The *required operations* refer to the set operation being performed, and these are similar to the corresponding operations described under *Algorithms for Small Data Volumes*.

For the purpose of comparing the two methods, let us assume that there are b bins and N records in both the lists. Let m be the number of records in each bin and n be the number of records in each CADM.

For this method, the order of complexity of sort is between O(N) and $O(N^2)$, depending upon the input data. For the post-sort phase, we can derive the order of complexity as follows :

Number of search operations to be performed = $bN = N^2/m$.

Order of complexity = $O((N^2/m)\log n)$

A second method(Method II) can be performed as follows. Both the lists are sorted using the CADM. Then the following operations are performed :

Method II :

```
Step1 : b = # bins in list A. (Size of list A / CADM Size)
Step2 : numrecs = number of records in list B.
Step3 : i = 0
    For j = 0 to b DO
            Load $Bin_j$ of list A into CADM.
            DONE = FALSE.
            While i < numrecs && DONE = FALSE  DO
                    Get $Record_i$ from list B.
                    Perform Search for $Record_i$ in CADM.
            Perform *required operations*.
            If END of CADM,  DONE = TRUE.
            i = i+1
        End DO
    End FOR
```

For this method, the order of complexity of sort is between O(2N) and O($2N^2$), depending upon the input data. For the post-sort phase, we can derive the order of complexity as follows :

Number of search operations to be performed = N

Order of complexity = O(N log n)

It is evident that the second method is better because of the following reasons :

- The order of complexity of post-sort phase for Method II (O(N log n)) is much better than that of Method I (O((N^2/m)log n)).
- The result will be in sorted order for method II.

This observation holds good if the sorting phase is fast enough. Since the sorting using CADM is fast, Method II has been employed.

3.4.31 Union :

Two methods were considered here. The Method I closely follows the algorithm used for small data volumes in that the records from list-A, which are eligible to be in the result are inserted in the right place in list-A after the search. In Method II, such records are kept in a separate list and are appended to the list-A at the end. The resulting list has to be resorted the get the result in sorted order.

Union Method I :

Let the lists whose union is to be found, be A and B.

```
Step1 : Sort Lists A and B using CADM and put the result back into main memory.

Step2 : j = (# records in CADM / 2)

Step3 : numrecs_a = number of records in list A.

        numrecs_b = number of records in list B.

Step4 : i = 0

        k = 0

        While k < numrecs_a DO

                Load j records from Top of list A into CADM.
        Top of list A = Top of list A + j
        DONE = FALSE.
        While i < numrecs_b && DONE = FALSE  DO
                Get Record$_i$ from list B.
                Perform *Search* for Record$_i$ in CADM.
                if Record$_i$  not present in CADM
                            PUSH Record$_i$ into CADM (list A).
                If END of CADM,  DONE = TRUE.
                i = i+1
        End DO
        Append CADM contents up to the address pointer to answer
buffer.
        j = (# records in CADM / 2) - (# records remaining in CADM)
        k = k+1
End DO
        Append rest of list A to answer buffer.
```

Note that only half the CADM space is filled at the beginning of each step. This is done to allow records to be pushed in. At the end, the Answer Buffer contains the Union of lists A and B.

Union Method II :

Let the lists whose union is to be found, be A and B.

```
Step1 : Sort Lists A and B using CADM and put the result back into main memory.

Step2 : $b' = \#$ bins in list A. (Size of list A / (2 * CADM Size))

Step3 : numrecs = number of records in list B.

Step4 : i = 0

    For j = 0 to $b'$ DO

        Load $Bin_j$ of list A into CADM.

        DONE = FALSE.

        While i < numrecs && DONE = FALSE  DO

            Get $Record_i$ from list B.

      Perform *Search* for $Record_i$ in CADM.

            if $Record_i$ not present in CADM

                Append $Record_i$ to Answer buffer.

            If END of CADM,  DONE = TRUE.

            i = i+1

        End DO

        j = j+1

    End FOR
```

Step5 : Append List A to the answer buffer and sort the result.

A comparison of Method I with Method II is given in the chapter 4.

3.4.32 Intersection :

Let the lists whose intersection is to be found, be A and B.

Step1 : Sort Lists A and B using CADM and put the result back into main memory.

Step2 : b = # bins in list A. (Size of list A / CADM Size)

Step3 : numrecs = number of records in list B.

```
Step4 : i = 0
    For j = 0 to b DO
            Load Bin_j of list A into CADM.
            DONE = FALSE.
            While i < numrecs && DONE = FALSE  DO
                    Get Record_i from list B.
                    Perform Search for Record_i in CADM.
                    if Record_i  present in CADM
                                Put Record_i into Output Buffer.
                    i = i+1
                    If END of CADM,  DONE = TRUE.
```

```
        End DO
        j = j+1
    End FOR
```

The output buffer now contains Intersection of A and B.

3.4.33 Difference :

Let the lists whose difference is to be found, be A and B.

Step1 : Sort Lists A and B using CADM and put the result back into main memory.

Step2 : b = # bins in list A. (Size of list A / CADM Size)

Step3 : numrecs = number of records in list B.

Step4 : i = 0

```
    For j = 0 to b DO
        Load $Bin_j$ of list A into CADM.
        DONE = FALSE.
        While i < numrecs && DONE = FALSE  DO
            Get $Record_i$ from list B.
            Perform Search for $Record_i$ in CADM.
            if $Record_i$  present in CADM
```

```
                    POP Record$_i$ from CADM (list A).
                i = i+1
                If END of CADM,  DONE = TRUE.
            End DO
        End FOR
```

Now, CADM array A contains (A-B).

3.5 RAM REQUIREMENTS FOR THE ALGORITHMS :

RAM memory requirements for the algorithms are an important consideration in the design of the Sort engine. It is necessary to keep the requirement as low as possible, to keep the cost of the system down.

3.5.1 Memory requirements of Algorithms for small data volumes :

Sorting, Union and *Difference* operations do not require any additional RAM other than the CADM buffer space. *Intersection* operation requires RAM memory of size equal to the size of the smaller of the two lists, in the worst case. *Selection* operation requires RAM memory equal to the size of the input list, in the worst case. The memory requirements depend on the selection criterion.

3.5.2 Memory requirements of Algorithms for large data volumes :

The Sorting operation requires RAM memory of size equal to twice the size of the list being sorted. For our Sort Engine, a RAM memory of 1.2 MBytes is required. This essentially forms the requirement for all other algorithms. For operations involving two lists, RAM memory of size equal to twice the sum of sizes of the two lists is required.

CHAPTER 4

SORT ENGINE SIMULATOR

An important issue in any engineering design is evaluating the alternatives available, before deciding on any particular design. The motive is to compare the alternative designs and pick the one that gives the best performance within the required cost. An important tool for this evaluation of designs of computer systems is a simulator that simulates the system under development. A simulator was developed to evaluate the performance and validate the algorithms for the Sort Engine. This simulator is capable of performing all Sort Engine operations and reports the time required to perform each of these operations. The routines which perform and time the *Off-Line Sort* and *Find* operations of the CADM were provided by Advanced Micro Devices Inc. Source code listing for all the other routines which were developed by the author are in Appendix B. Routines were developed to perform these operations entirely on the SUN 3/50, for validation and comparison. The source code listing for these routines are also included in Appendix B.

4.1 STRUCTURE OF THE SIMULATOR :

The structure of this simulator is shown in Figure 13. The *Command*

Scanner reads the command line from the user and initializes various parameters such as key length, pointer length, the operation being performed, the system to be used (SUN 3/50 or CADM Sort Engine), input file names, etc. Then, depending upon the operation being performed, the required routine is called by the *Routine Selector*. After the operation is completed, the *Output* routine writes out the result of the operation and the statistics, which contains the timing information along with other things shown below. The output is written to a file *result.out* and the statistics are written to the file *result.stat* in the current directory.

4.2 Algorithms for performing operations on SUN 3/50 :

The algorithms for performing sorting and various set operations on the SUN 3/50 are described in the Appendix A. These are very similar to the algorithms used in the Sort Engine, but altered slightly to make them more efficient. Operations such as the POP and PUSH in CADM, are very time consuming on conventional machines. Such operations are replaced.

```
CADM SORT ENGINE SIMULATOR STATISTICS

        Operation :          Union

        File_A :             rand4
```

```
# Records_A : 930

# Records_B : 882

Key length :    12

Pointer length :          4

# Sort Clocks :431032

# Clock cycles :         378054

Sort time :       26.939 millisecs

Total time :      50.568 millisecs
```

An example of simulator output format

4.3.1 Results for Small data volumes :

Off-Line sort in the CADM is data dependent. As shown in Figure 14, best results are obtained for pre-sorted data. Order of complexity is linear here. The worst case performance for reverse sorted data, shows that the order of complexity is quadratic. Random data falls in between the two. As shown by Figure 15, sorting in CADM is about 20 times faster than Quicksort on SUN 3/50. This was for random data.

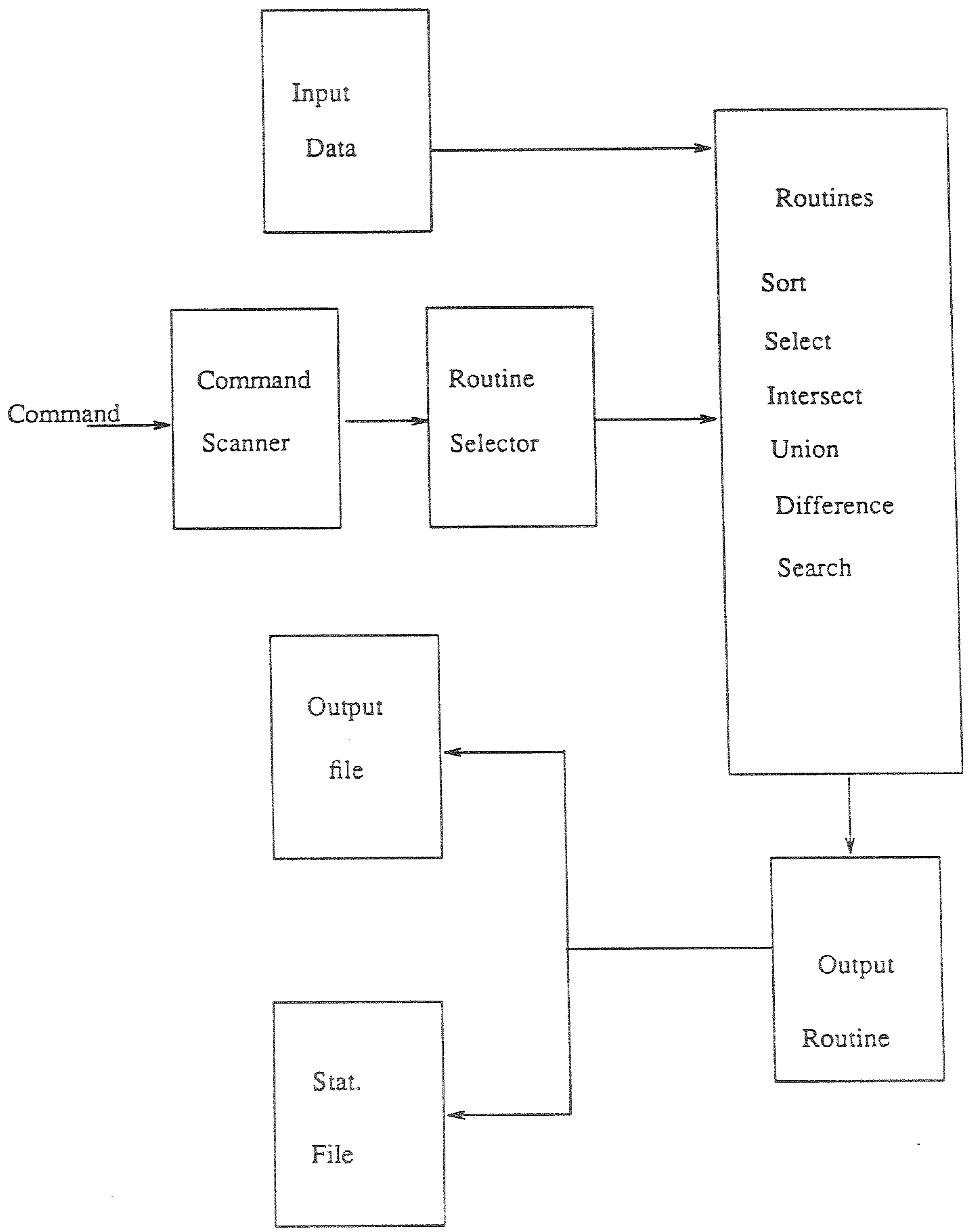


Figure 13. Structure of the simulator

For set operations, CADM proved to be about 20 times faster than SUN 3/50. As shown in Figures 16, 17 and 18, the plots for SUN 3/50 vary depending on the operation performed. Since the CADM time is small, corresponding plots for CADM do not show any appreciable change. In all these plots, size of the List-B was kept constant at 15KBytes.

4.3.1 Results for Large data volumes :

In any large Database system, the data volumes handled are in this category, which makes these results very important. The first issue is the performance of sort. Figure 19 shows the performance of Algorithms A, B and C described in section 3.4.1, along with the performance of Quicksort on SUN 3/50. It is evident that Algorithm C out performs all the others. Hence, Algorithm C was chosen for sorting in the sort engine. A single CADM bank of 16 CADMs was used for this comparison.

The next important issue is the CADM configuration to be used. The performance of Multiple bank Configuration-A (Figure 11a in section 3.1) was compared to the single bank Configuration-B (Figure 11b in section 3.1). Figure 20 shows that Configuration-A outperforms Configuration-B. Hence, this

SMALL DATA VOLUMES

DATA DEPENDENCE OF CADM SORT

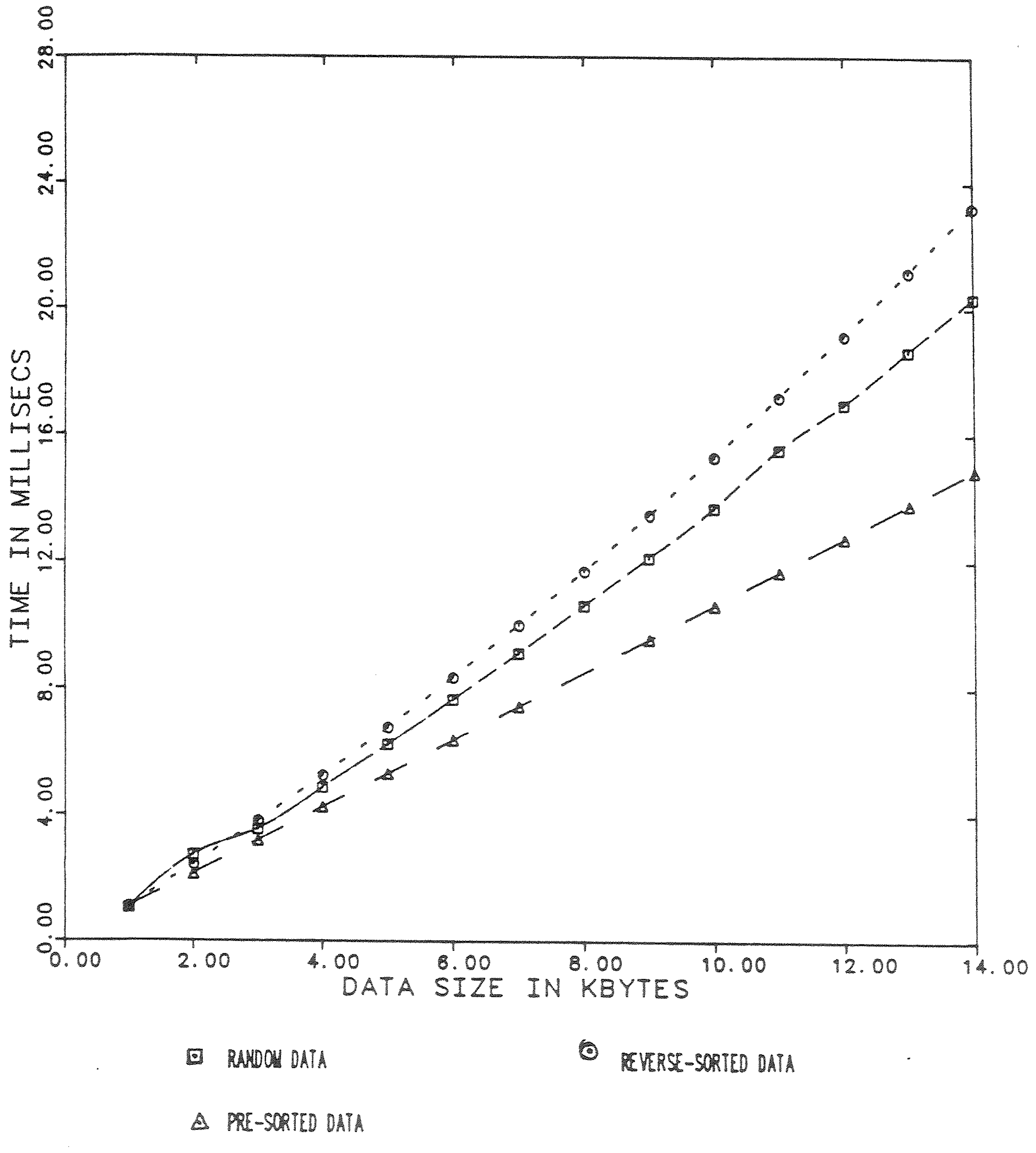


Figure 14

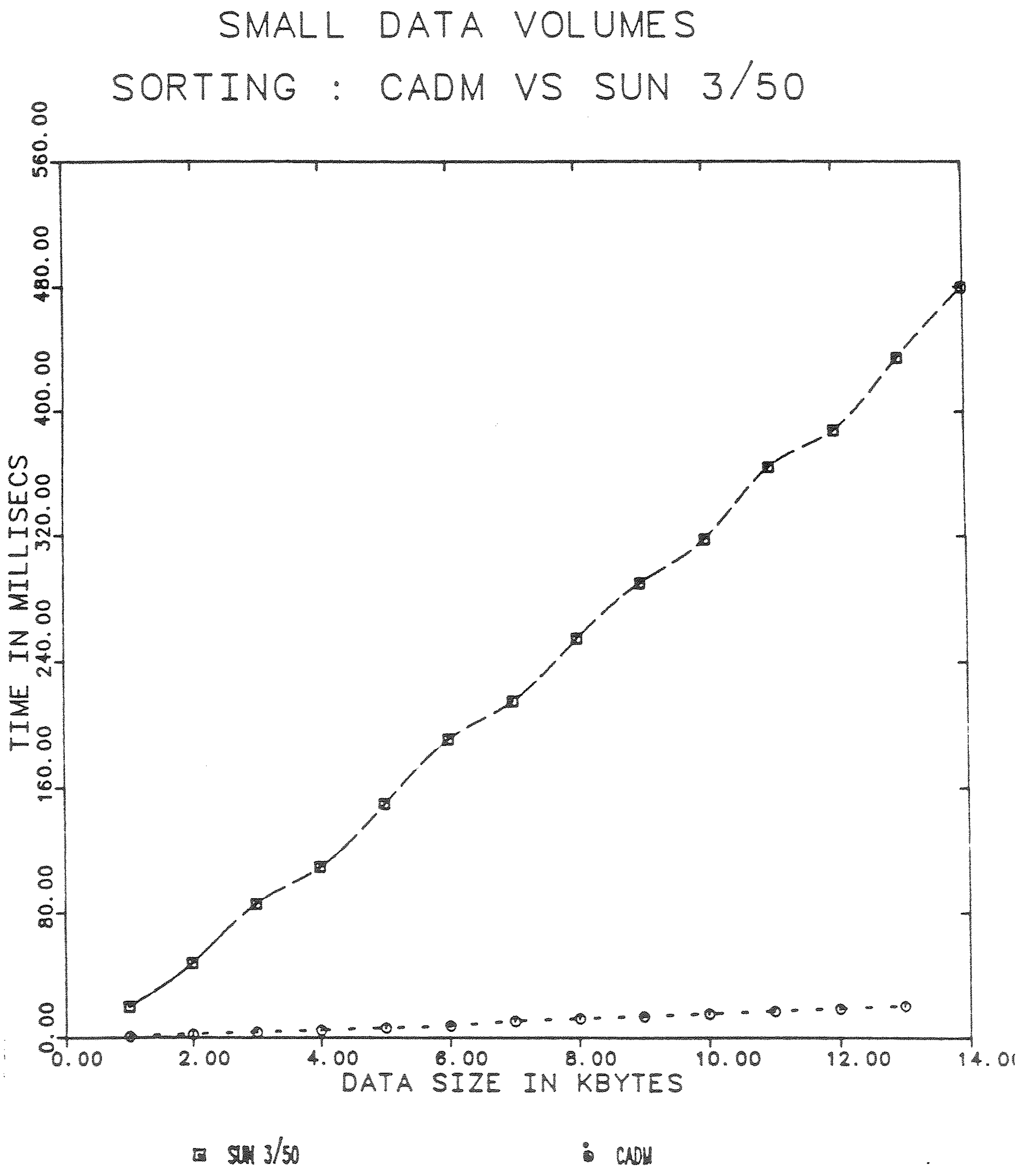

SMALL DATA VOLUMES
SORTING : CADM VS SUN 3/50
TIME IN MILLISECS
0.00
80.00
160.00
240.00
320.00
400.00
480.00
560.00
DATA SIZE IN KBYTES
0.00
2.00
4.00
6.00
8.00
10.00
12.00
14.0
SUN 3/50
CADM

Figure 15

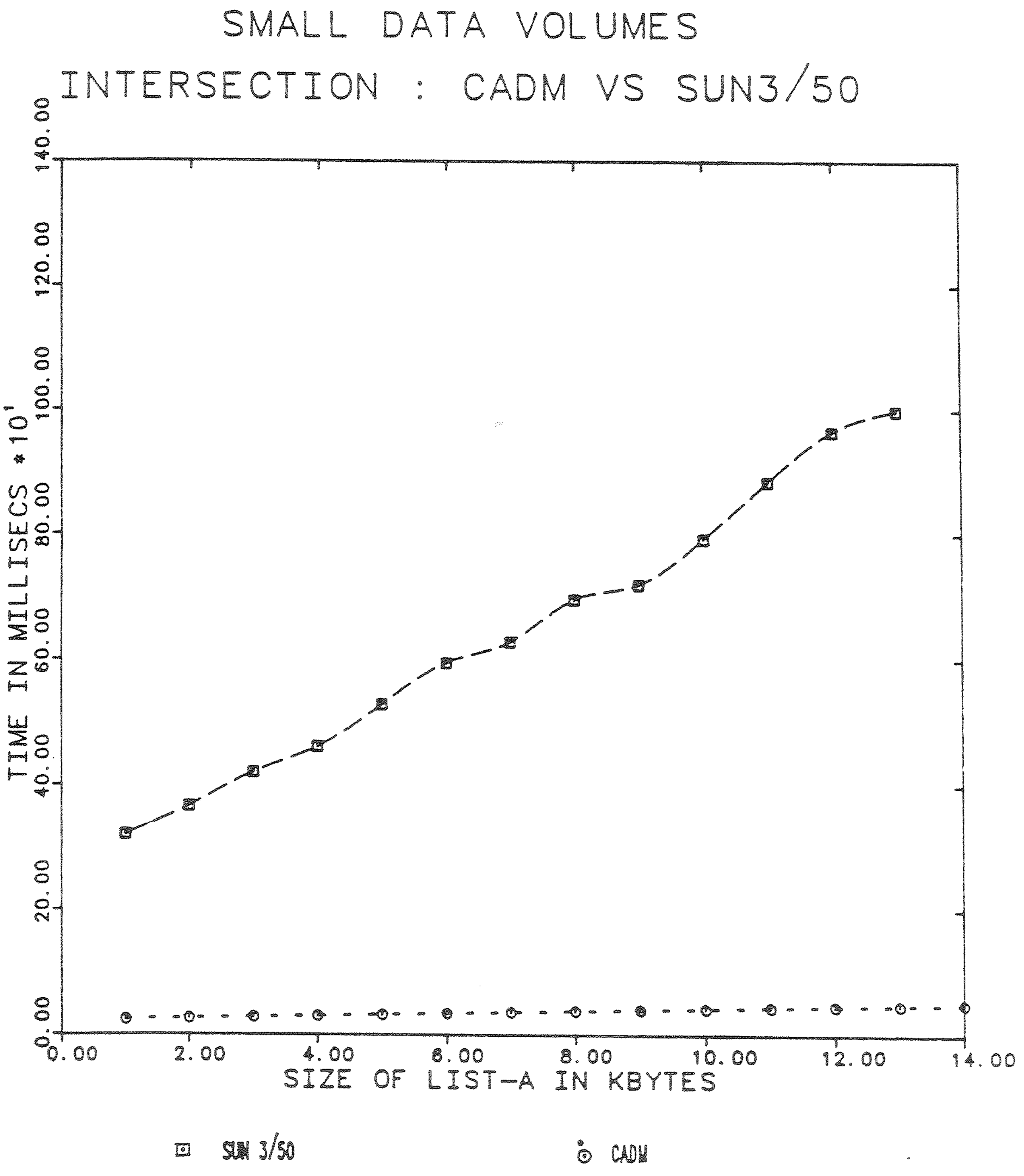

SMALL DATA VOLUMES
INTERSECTION : CADM VS SUN3/50
TIME IN MILLISECS *10'
SIZE OF LIST-A IN KBYTES
0.00
20.00
40.00
60.00
80.00
100.00
120.00
140.00
0.00
2.00
4.00
6.00
8.00
10.00
12.00
14.00
SUN 3/50
CADM

Figure 16

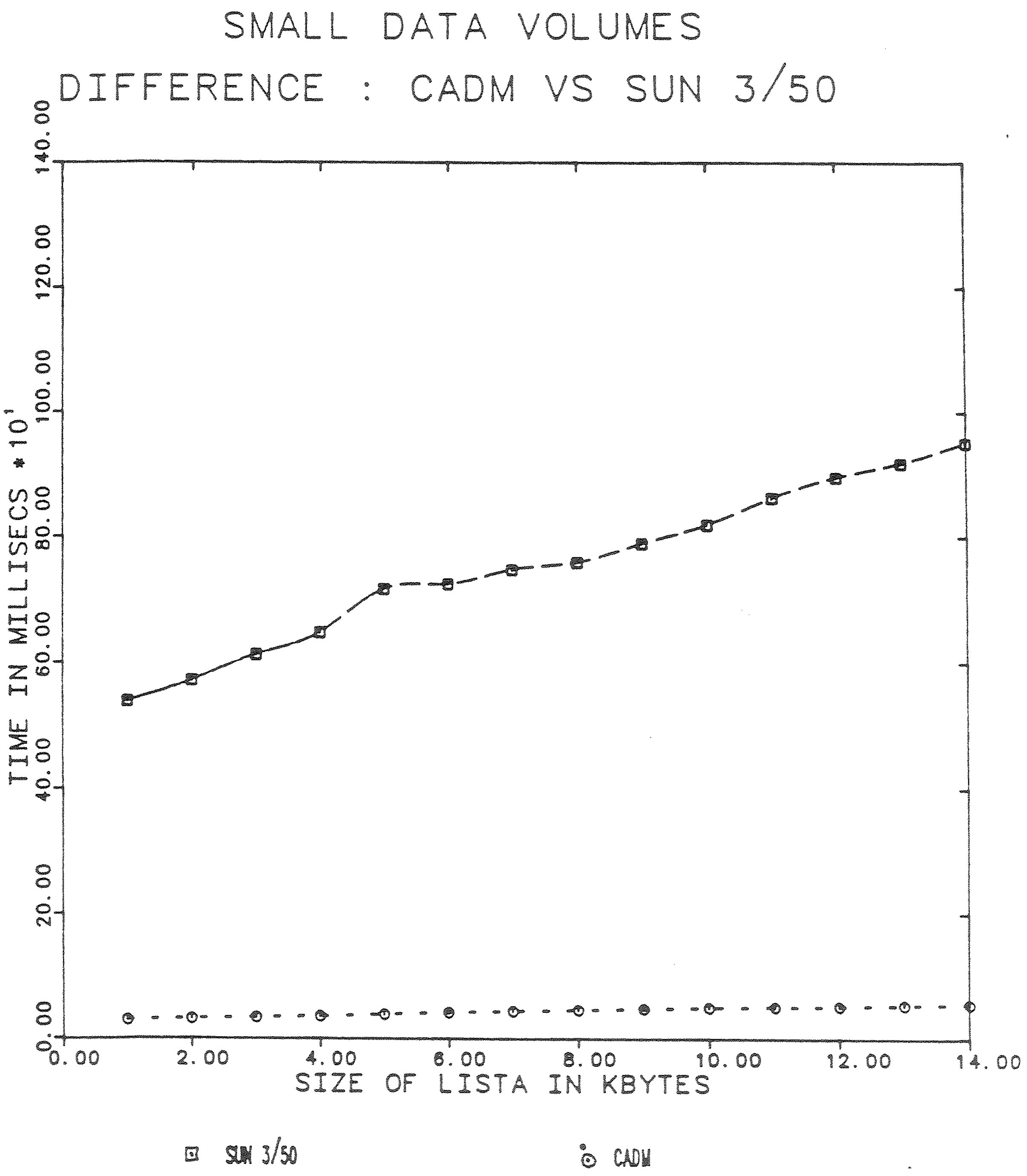

SMALL DATA VOLUMES
DIFFERENCE : CADM VS SUN 3/50
TIME IN MILLISECS *10¹
SIZE OF LISTA IN KBYTES
0.00
20.00
40.00
60.00
80.00
100.00
120.00
140.00
2.00
4.00
6.00
8.00
10.00
12.00
14.00
SUN 3/50
CADM

Figure 17

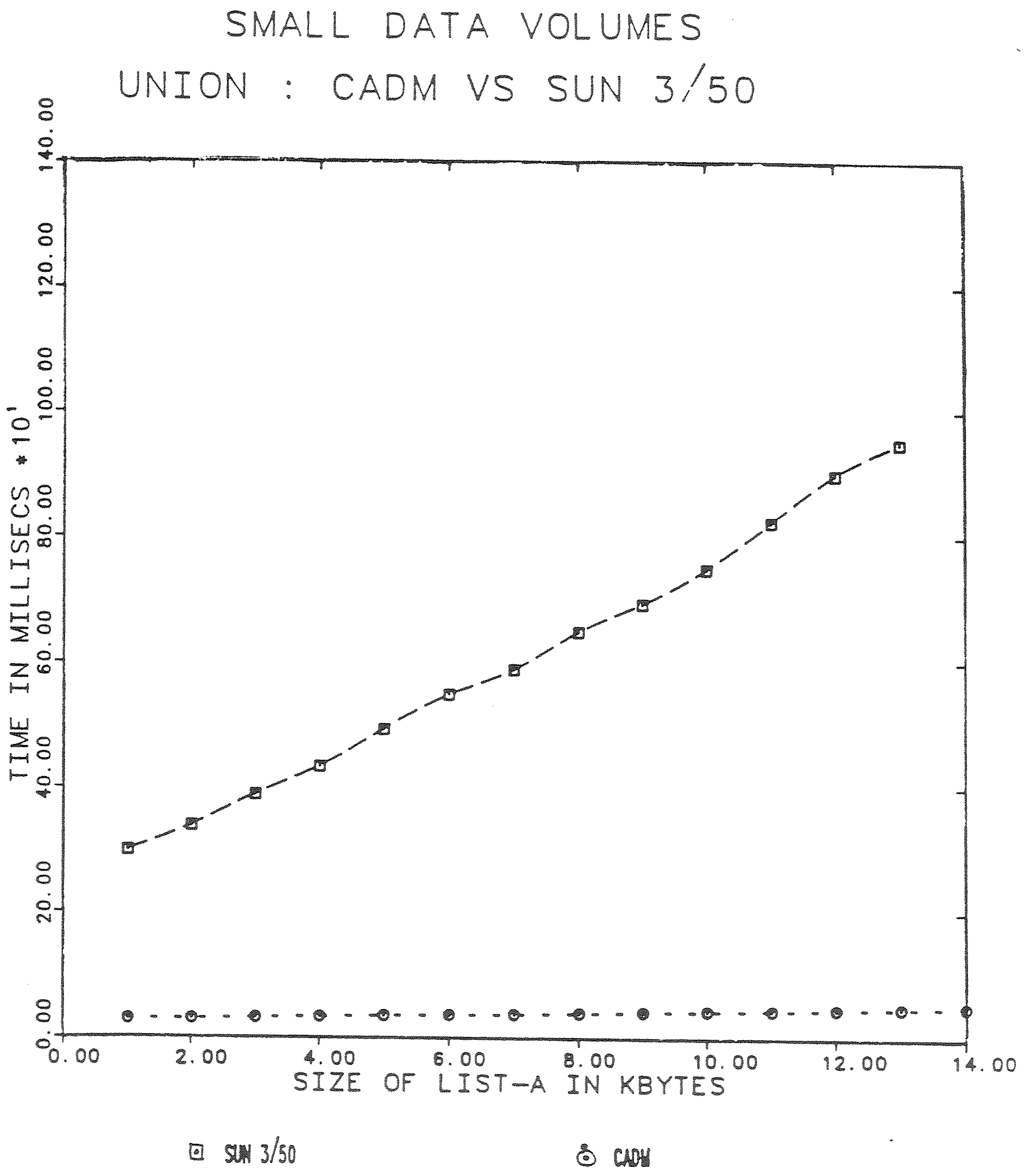

SMALL DATA VOLUMES
UNION : CADM VS SUN 3/50
TIME IN MILLISECS *10¹
0.00
20.00
40.00
60.00
80.00
100.00
120.00
140.00
0.00
2.00
4.00
6.00
8.00
10.00
12.00
14.00
SIZE OF LIST-A IN KBYTES
SUN 3/50
CADM

Figure 18

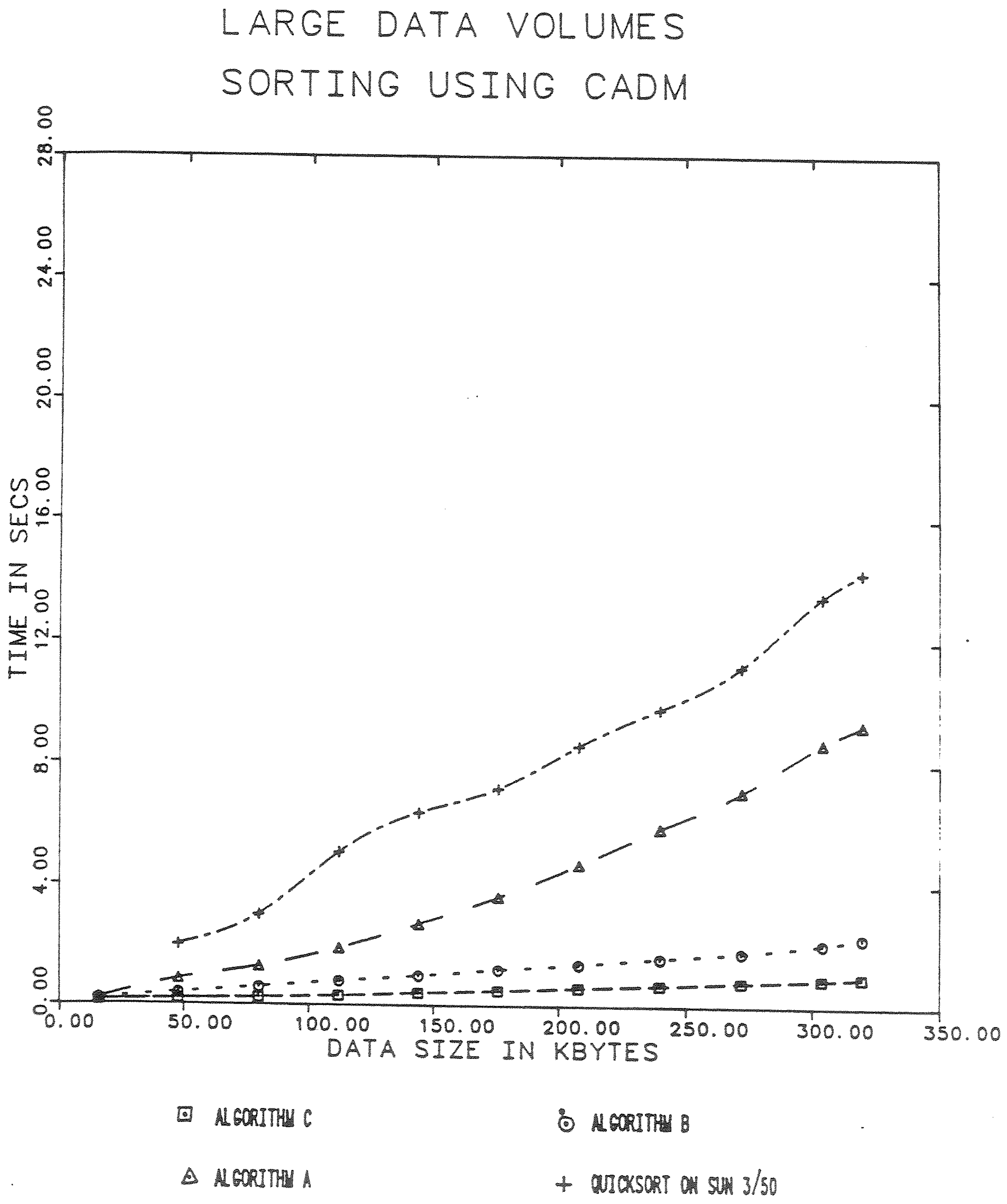

LARGE DATA VOLUMES
SORTING USING CADM
TIME IN SECS
0.00
4.00
8.00
12.00
16.00
20.00
24.00
28.00
DATA SIZE IN KBYTES
0.00
50.00
100.00
150.00
200.00
250.00
300.00
350.00
ALGORITHM C
ALGORITHM B
ALGORITHM A
QUICKSORT ON SUN 3/50

Figure 19

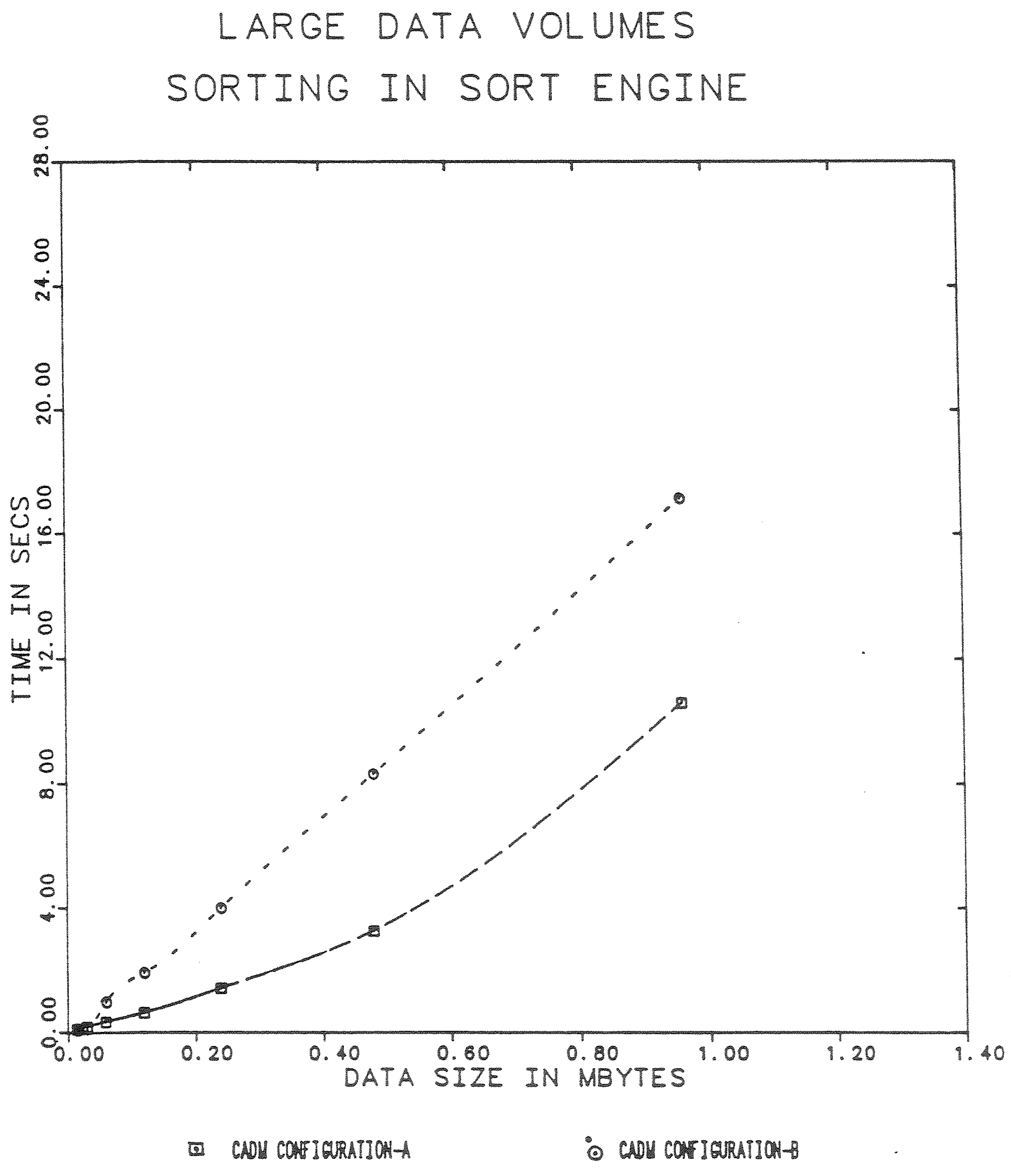
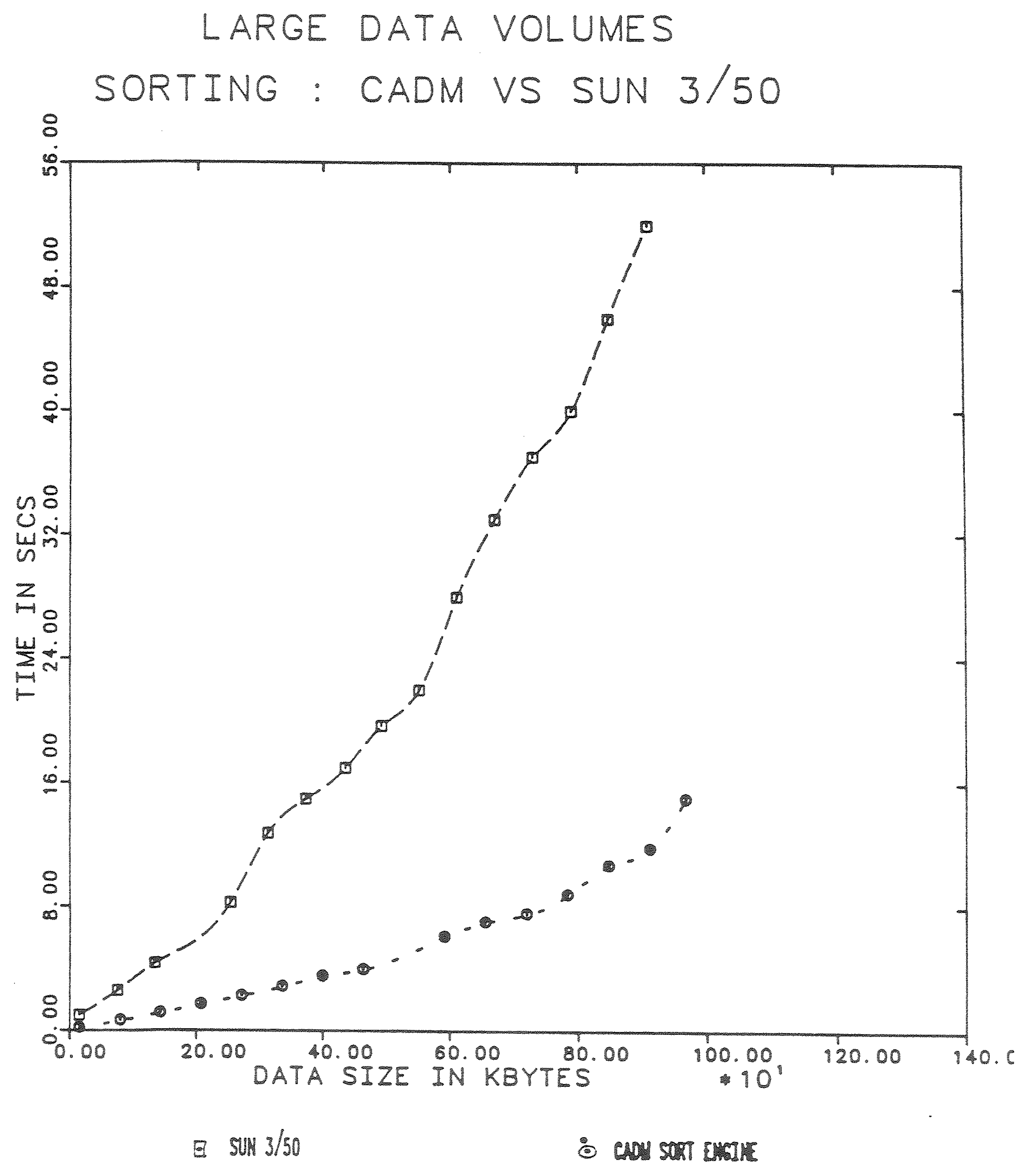

LARGE DATA VOLUMES
SORTING IN SORT ENGINE
TIME IN SECS
DATA SIZE IN MBYTES
0.00
4.00
8.00
12.00
16.00
20.00
24.00
28.00
0.00
0.20
0.40
0.60
0.80
1.00
1.20
1.40
CADM CONFIGURATION-A
CADM CONFIGURATION-B

Figure 20



LARGE DATA VOLUMES
SORTING : CADM VS SUN 3/50
TIME IN SECS
0.00
8.00
16.00
24.00
32.00
40.00
48.00
56.00
0.00
20.00
40.00
60.00
80.00
100.00
120.00
140.
DATA SIZE IN KBYTES
*10¹
SUN 3/50
CADM SORT ENGINE

Figure 21

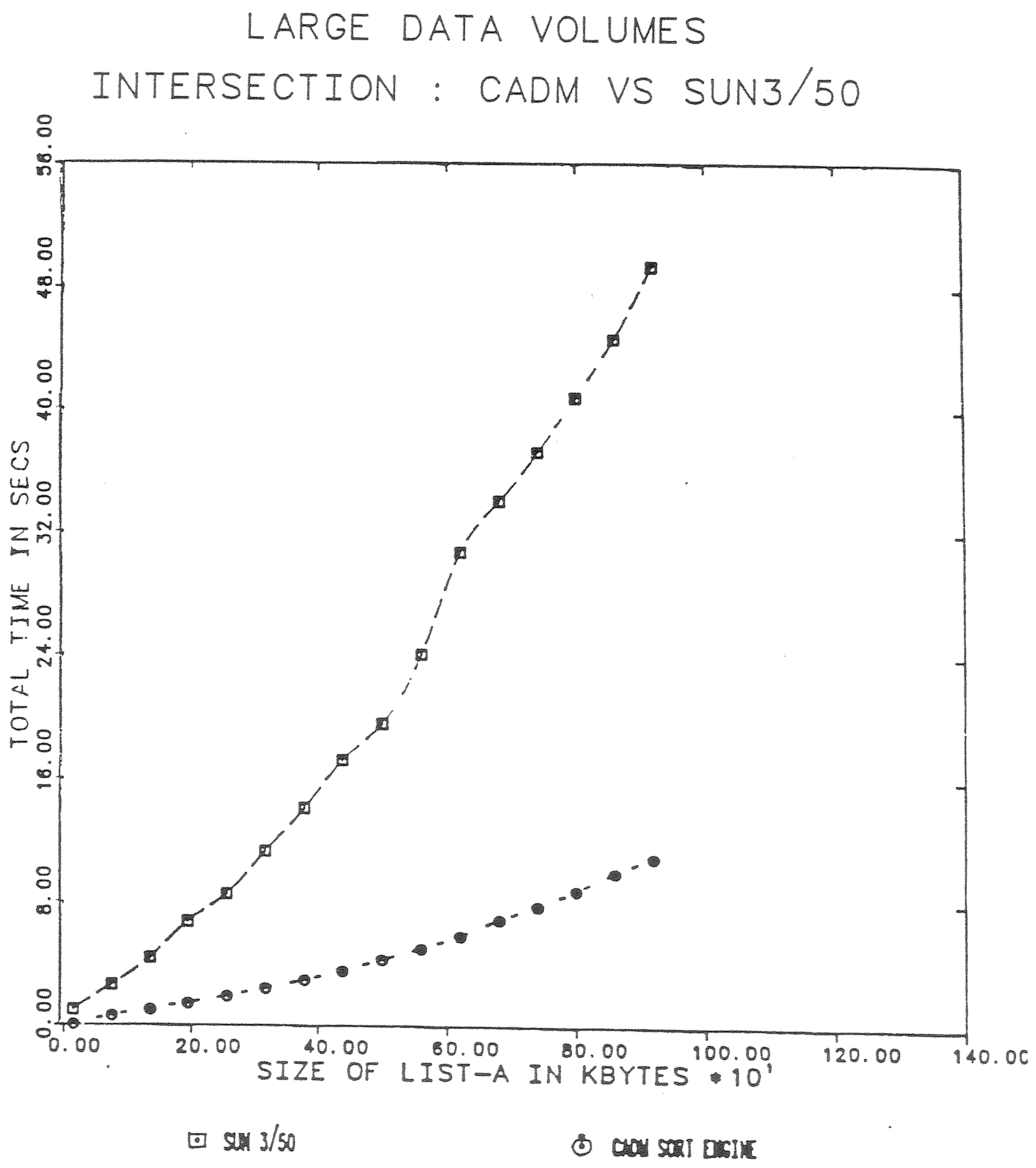

LARGE DATA VOLUMES
INTERSECTION : CADM VS SUN3/50
TOTAL TIME IN SECS
0.00
8.00
16.00
24.00
32.00
40.00
48.00
56.00
0.00
20.00
40.00
60.00
80.00
100.00
120.00
140.00
SIZE OF LIST-A IN KBYTES *10'
SUN 3/50
CADM SORT ENGINE

Figure 22

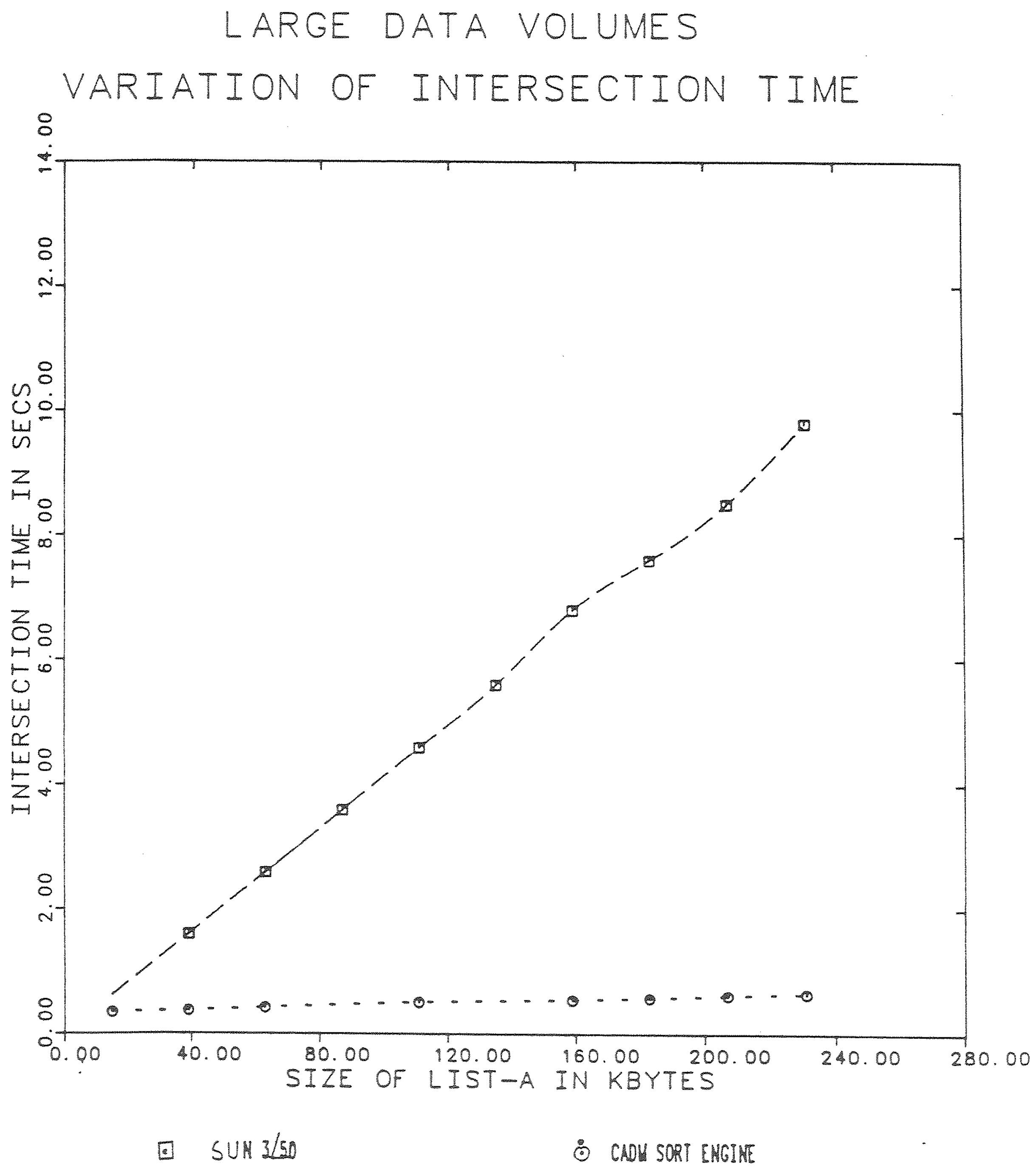
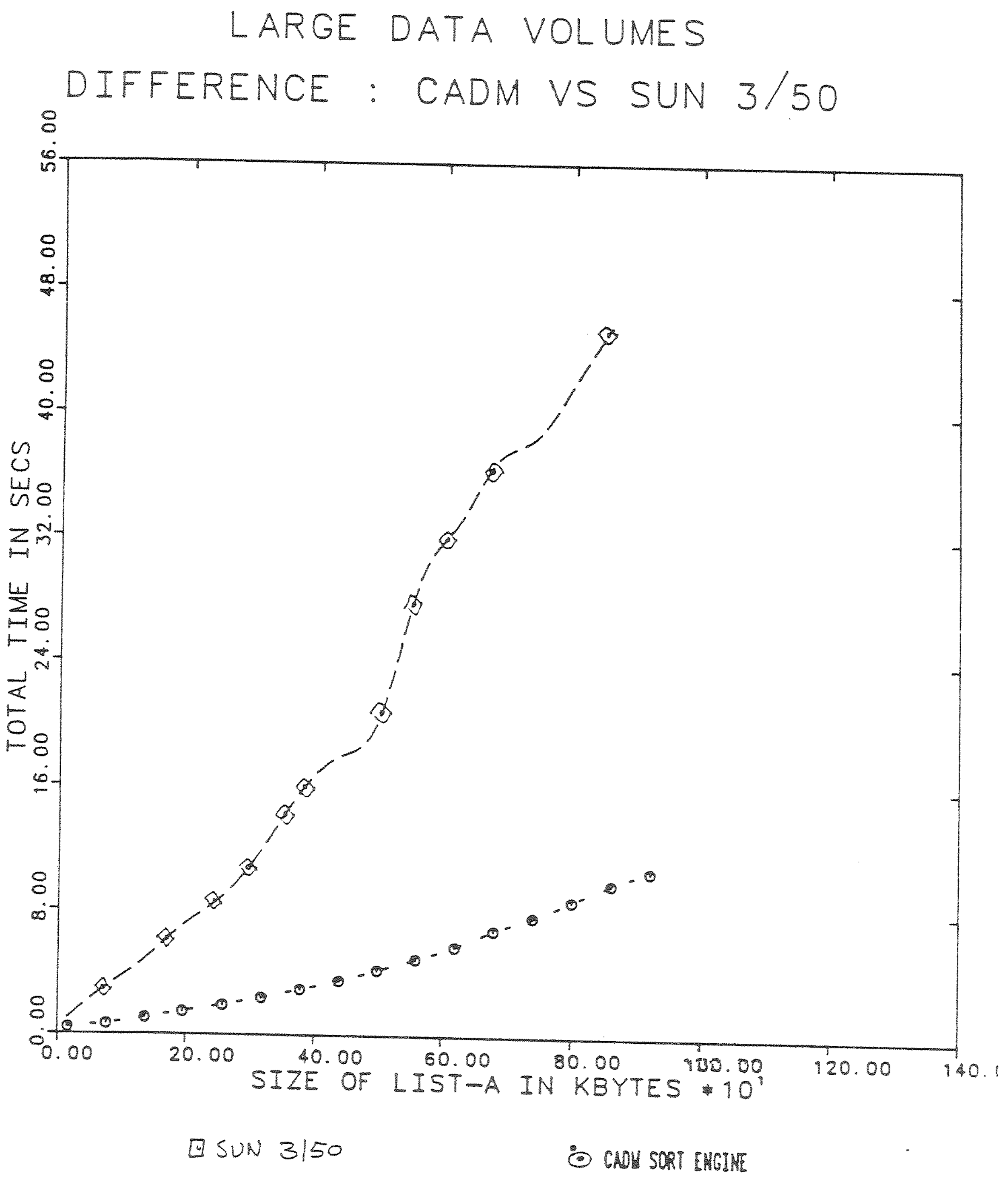
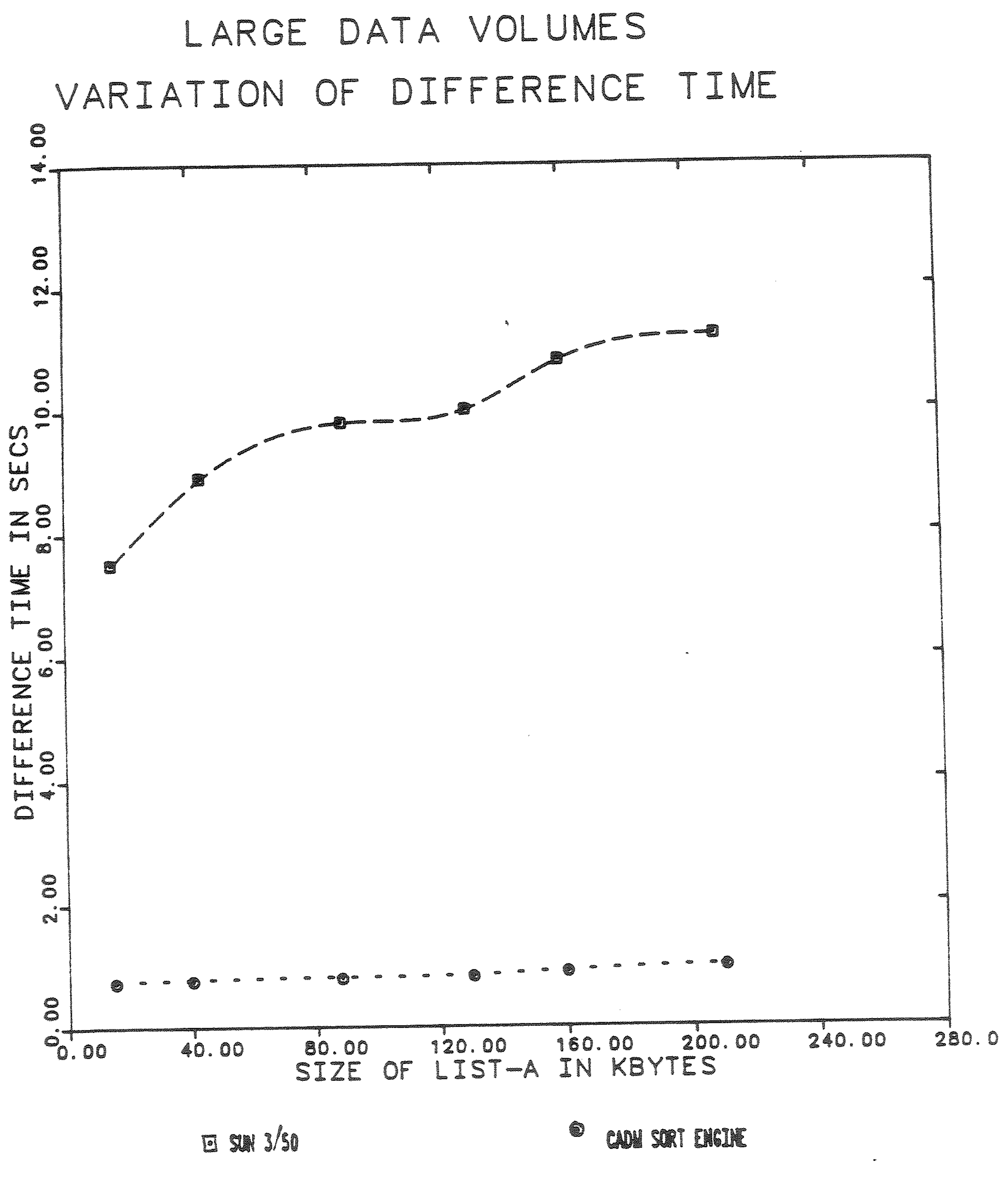

LARGE DATA VOLUMES
VARIATION OF INTERSECTION TIME
INTERSECTION TIME IN SECS
0.00
2.00
4.00
6.00
8.00
10.00
12.00
14.00
0.00
40.00
80.00
120.00
160.00
200.00
240.00
280.00
SIZE OF LIST-A IN KBYTES
SUN 3/50
CADM SORT ENGINE

Figure 23



LARGE DATA VOLUMES
DIFFERENCE : CADM VS SUN 3/50
TOTAL TIME IN SECS
0.00
8.00
16.00
24.00
32.00
40.00
48.00
56.00
0.00
20.00
40.00
60.00
80.00
100.00
120.00
140.
SIZE OF LIST-A IN KBYTES * 10¹
SUN 3/50
CADM SORT ENGINE

Figure 24



LARGE DATA VOLUMES
VARIATION OF DIFFERENCE TIME
DIFFERENCE TIME IN SECS
0.00
2.00
4.00
6.00
8.00
10.00
12.00
14.00
0.00
40.00
80.00
120.00
160.00
200.00
240.00
280.0
SIZE OF LIST-A IN KBYTES
SUN 3/50
CADM SORT ENGINE

Figure 25

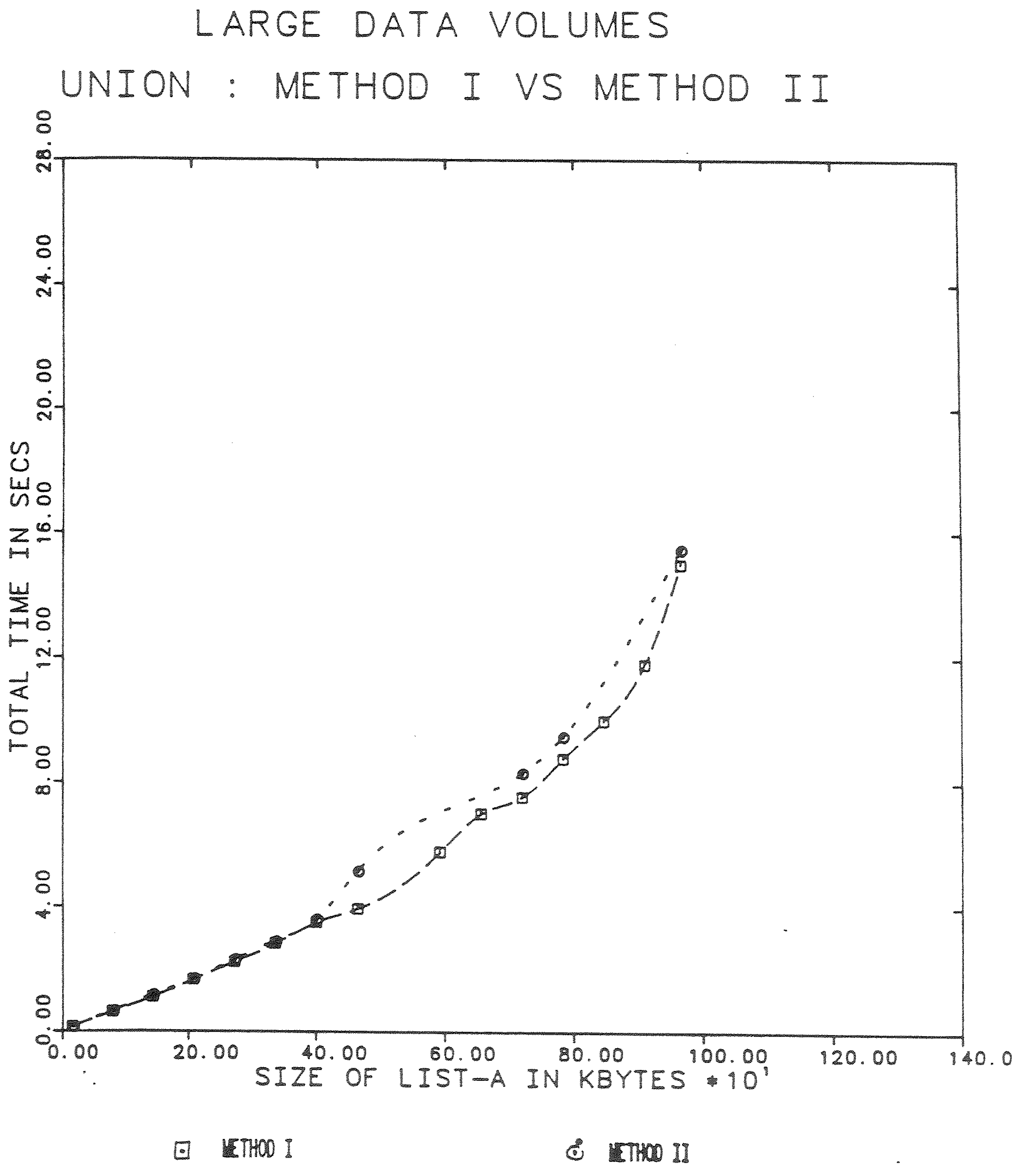
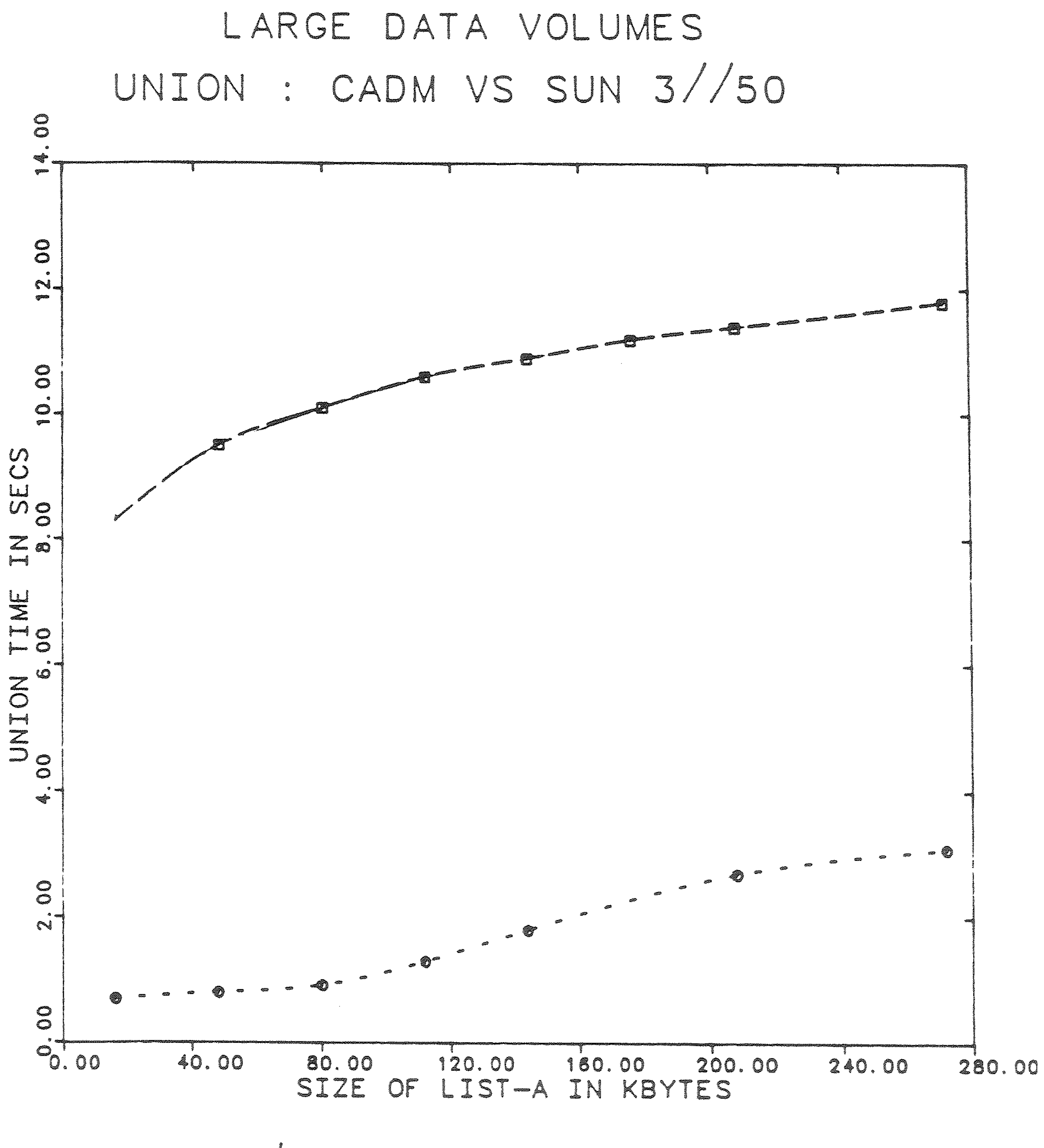

LARGE DATA VOLUMES
UNION : METHOD I VS METHOD II
TOTAL TIME IN SECS
0.00
4.00
8.00
12.00
16.00
20.00
24.00
28.00
0.00
20.00
40.00
60.00
80.00
100.00
120.00
140.0
SIZE OF LIST-A IN KBYTES *10'
METHOD I
METHOD II

Figure 26



LARGE DATA VOLUMES
UNION : CADM VS SUN 3//50
UNION TIME IN SECS
0.00
2.00
4.00
6.00
8.00
10.00
12.00
14.00
0.00
40.00
80.00
120.00
160.00
200.00
240.00
280.00
SIZE OF LIST-A IN KBYTES
SUN 3/50
CADM SORT ENGINE

Figure 27

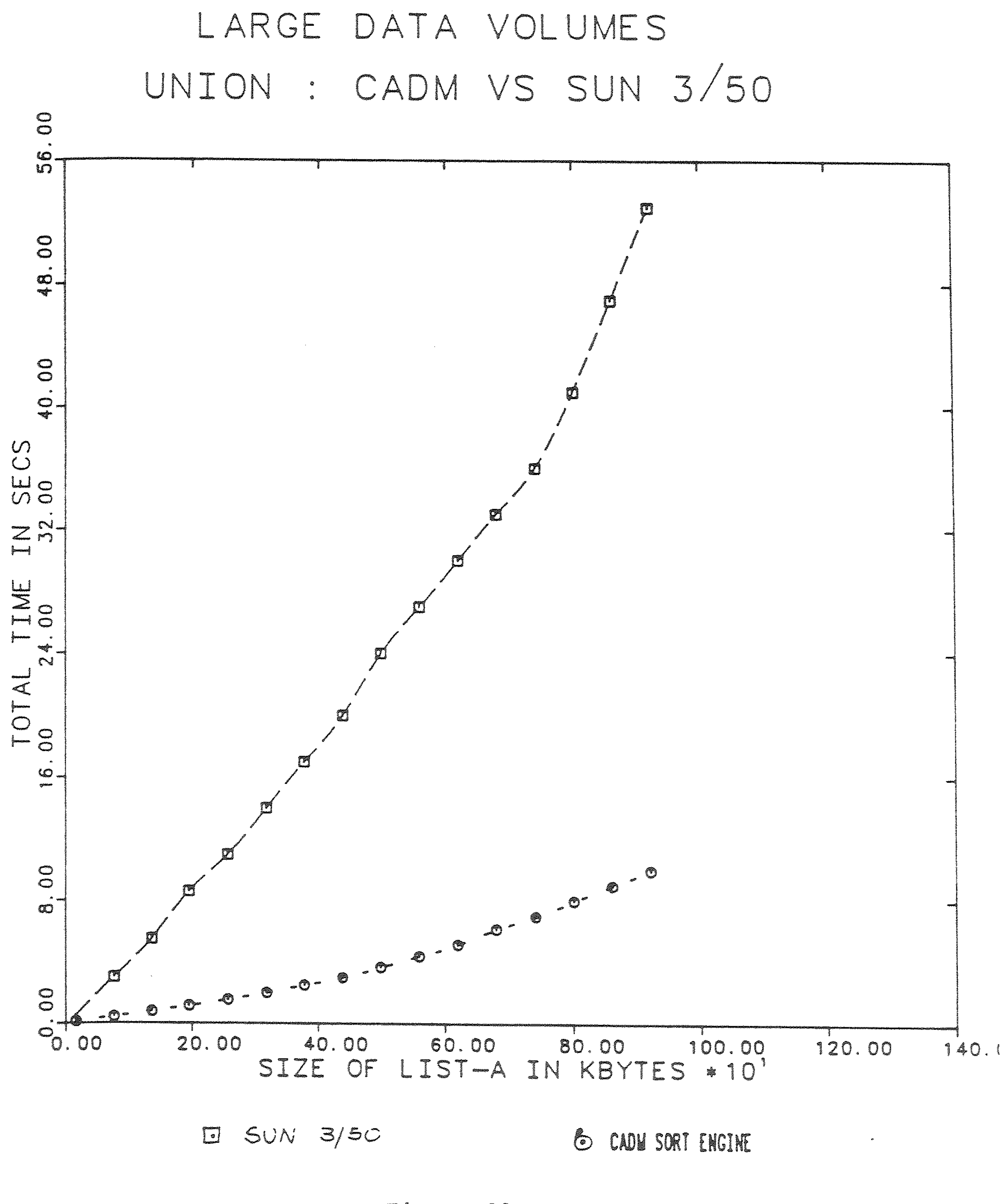

LARGE DATA VOLUMES
UNION : CADM VS SUN 3/50
TOTAL TIME IN SECS
0.00
8.00
16.00
24.00
32.00
40.00
48.00
56.00
0.00
20.00
40.00
60.00
80.00
100.00
120.00
140.
SIZE OF LIST-A IN KBYTES *10¹
SUN 3/50
CADM SORT ENGINE

Figure 28

configuration will be used. This configuration was used to get the results shown in the rest of the figures. The curve for Configuration-A shows the effect of a large number of bins for the portions of the graph where datasize is greater than 500 KBytes. At 1 MByte, there will be about 60 bins of 16 KBytes each. Figure 21 shows the performance of sort in Sort engine and on SUN 3/50. Sort engine is about eight times faster than SUN 3/50.

For set operations, Sort engine was about 5 times faster than SUN 3/50. This is shown in Figures 22, 24 and 27. It must also be noted that the operations performed after the initial sorting of the two lists, are also much faster in the Sort Engine than on SUN 3/50, as shown by Figures 23, 25 and 28. Of the two algorithms used for finding the Union for large data volumes, Method I is slightly superior to Method II. It must be noted that if result need not be in sorted order, Method II wins over Method I. This is evident from Figure 26.

CHAPTER 5

SORT ENGINE IMPLEMENTATION

One of the decisions in designing the Sort engine was to use as many off-the-shelf components as possible. Hence, the design incorporates VME modules for MC68020 processor, from Motorola. A prototype has been implemented, by interfacing the CADM to the TRS-80 Color Computer from Radio Shack. The reasons for using a TRS-80 were easy availability and ease of interfacing the hardware to it. The interface was through the Universal Project Board. The availability of a C compiler with optimizer, simplified the task of developing the required software. This setup is described in the following section. The design based on VME modules is described in the section after that. The implementation of the system based on VME modules is beyond the scope of this work. Hence, a basic design is suggested.

5.1 SORT ENGINE PROTOTYPE USING TRS-80 :

The CADM interface to TRS-80 [BHAT 86] included the associated hardware and device driver for OS-9, which is the operating system for TRS-80. The block diagram of the setup is shown in figure 29. The TRS-80, based on MC6809 microprocessor, communicates with the CADM hadware via an 8 bit parallel bus. A bi-directional bus driver, 74LS245 isolates the TRS-80 data bus from CADM data bus. Gadfly synchronization mechanism was used to synchronize the data transfer between the TRS-80 and CADM. The two output signals from CADM, *Done* and *Status* were connected to input ports on TRS-80. The address bus of TRS-80 was decoded to provide the chip-select (CS) signal for CADM. Address bit A_0 was used to select between command and data (C/D') for CADM. R/W' signal from TRS-80 was split up into read enable (RE) and write enable (WE) signals for the CADM. A standard clock generator was used to generate a clock of 20 MHz, which was divided by four with a counter (74LS193) to provide a 5 MHz clock for CADM. It is required that the input signals for CADM from the host change states only with the rising edge of CADM's clock for correct operation. Hence, synchronization circuits with two D flip-flops (74LS74) were used for C/D', RE, WE and CS signals. The schematic diagram of this interface is given in figures 30 and 31.

This implementation provided a good insight into the issues involved in the design of such a system. It was found that the CADM is extremely sensitive to the synchronization of its input signals with the rising edge of its clock. One of the undesirable features of the chip was also discovered. There is no instruction to increment the address pointer of CADM by any desired amount, other than of the key of that record. If only the pointer is to be read back, as many increment commands as the number of key bytes is to be executed to get to the pointer, which is a waste of time.

The prototype was built with the intention of validating the results of simulation as much as possible. The algorithms for small data volumes that were discussed before, were implemented and the results were checked. All the CADM instructions were exercised, and the number of clock cycles taken for each instruction were counted with the help of a logic analyzer. The table below compares the number of clock cycles from the simulation with the actual results obtained from the prototype.

The timings from the simulator agree well with the timings from the prototype. The off-line sort and find timings were measured with 63 records in CADM of 16 bytes each. The key size used was 12 bytes and the pointer size was 4 bytes. This validates the results for large data volumes from the simulator also, as the basic instruction timings used for simulation are accurate.

Operation	Simulator	Prototype
Write in AI Mode	6 Clock Cycles	6 Clock Cycles
Read in AI Mode	6 Clock Cycles	6 Clock Cycles
Write in Stack Mode	8 Clock Cycles	7 Clock Cycles
Read in Stack Mode	8 Clock Cycles	10 Clock Cycles
Find	220 Clock Cycles	156 Clock Cycles
Sort (Presorted data)	16008 Clock Cycles	15958 Clock Cycles
Sort (Random data)	16668 Clock Cycles	16158 Clock Cycles
Sort (Reverse Sorted data)	16841 Clock Cycles	16850 Clock Cycles

Table 2. Comparison of Simulator timing with actual timings from Prototype

5.2 SORT ENGINE WITH VME MODULES :

This design uses the MVME family of modules from Motorola Inc [MOTO 87]. The block diagram is shown in figure 31. The CADM module is the same as in figure 12. There are a total of four such CADM modules. The processor module is MVME133A, which has the MC68020 CPU with 2MBytes of RAM on board and 256 KBytes of ROM. The RAM can be accessed from the local bus on board, as well as from VME bus. The board also has DMA channel and required accessories. MVME025 System controller provides system clock, system reset, bus arbitration and bus timers. The clock generator on this controller can generate the clock for CADM modules. MVME940 chassis with 7 VME slots and power supply is used to house the system.

MVMEbug, a debugging package for VME modules, can be used to develop and test the system. This supports downloading of programs from another host through a serial port, in Motorola's S record format.

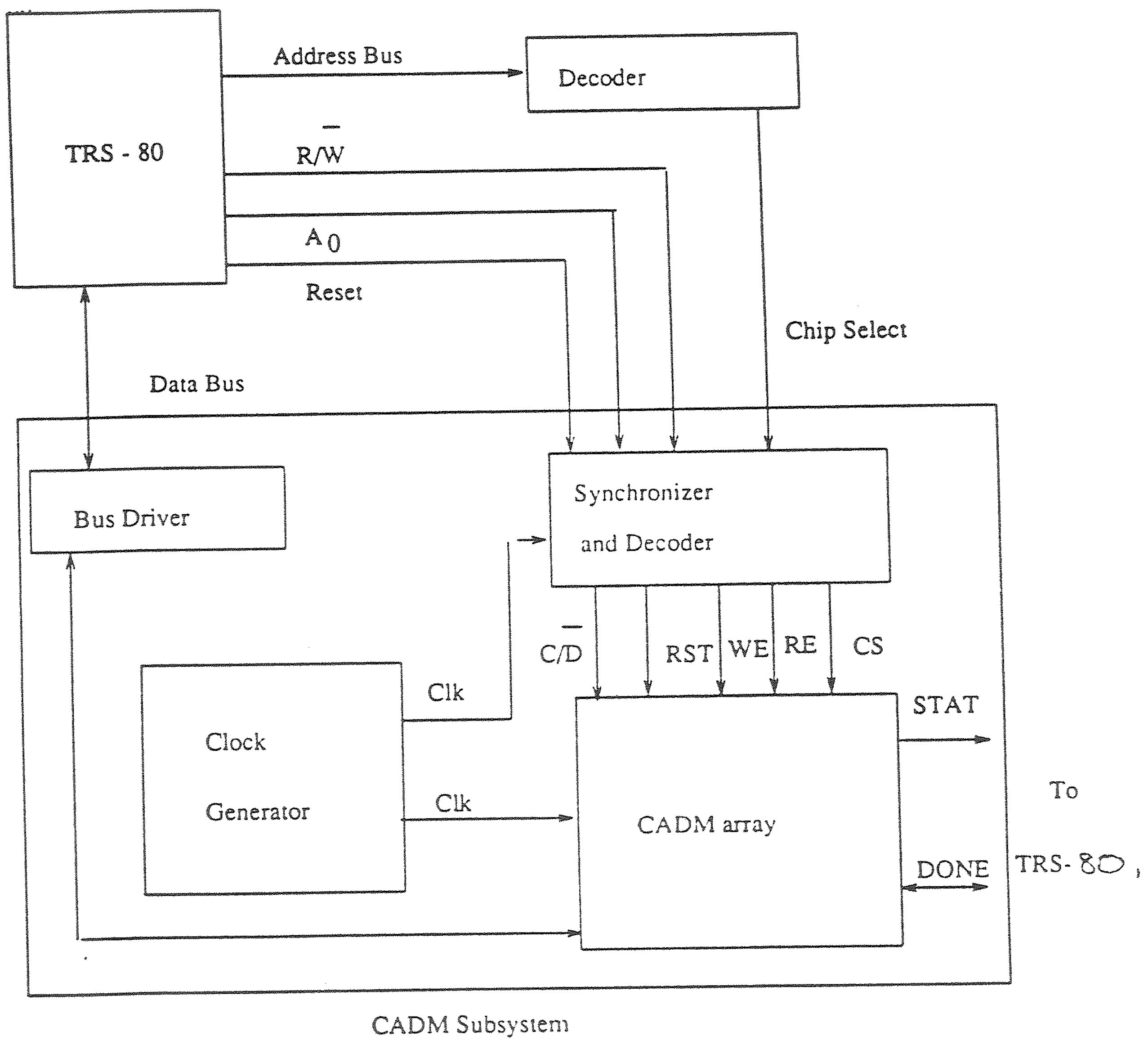
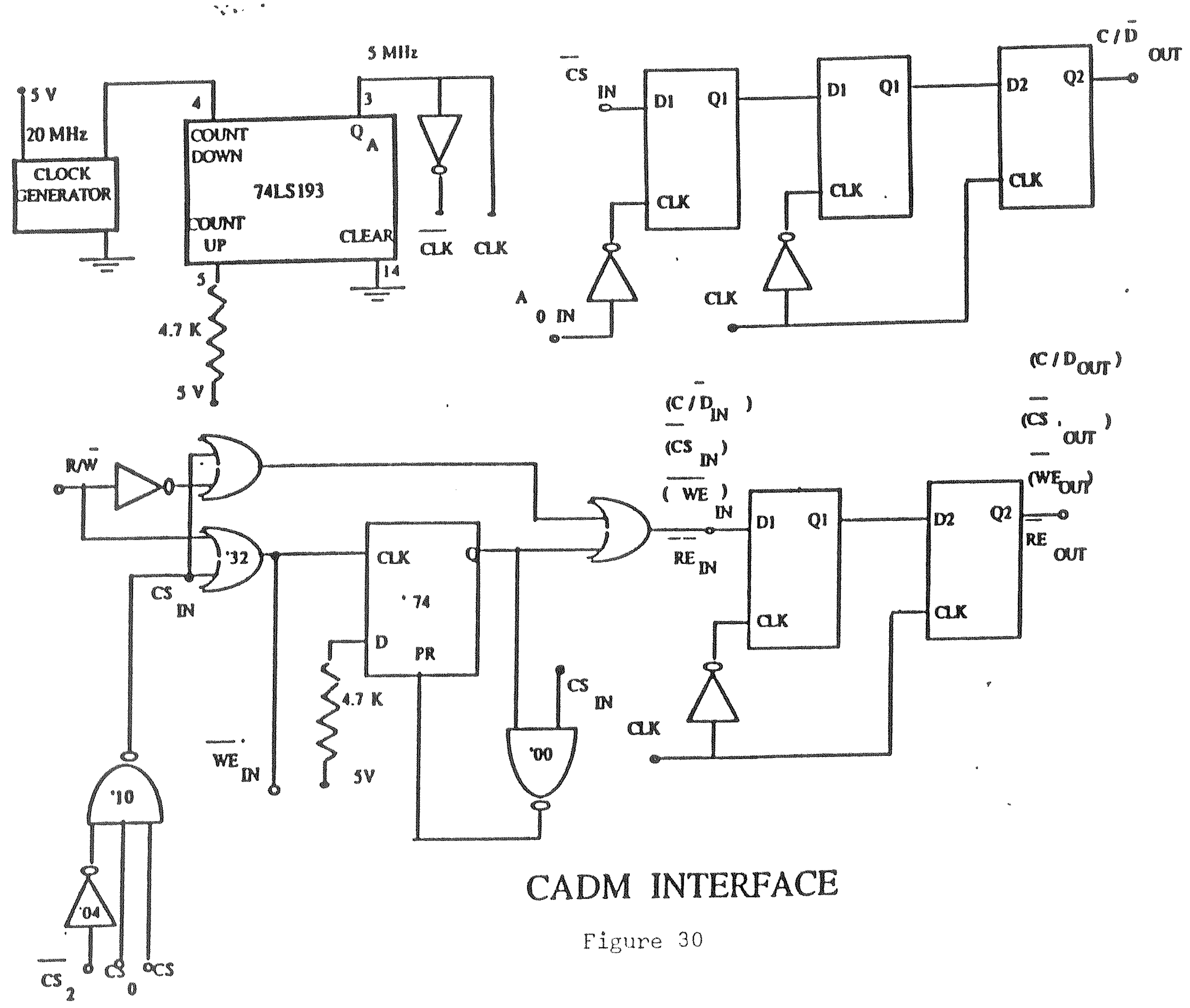
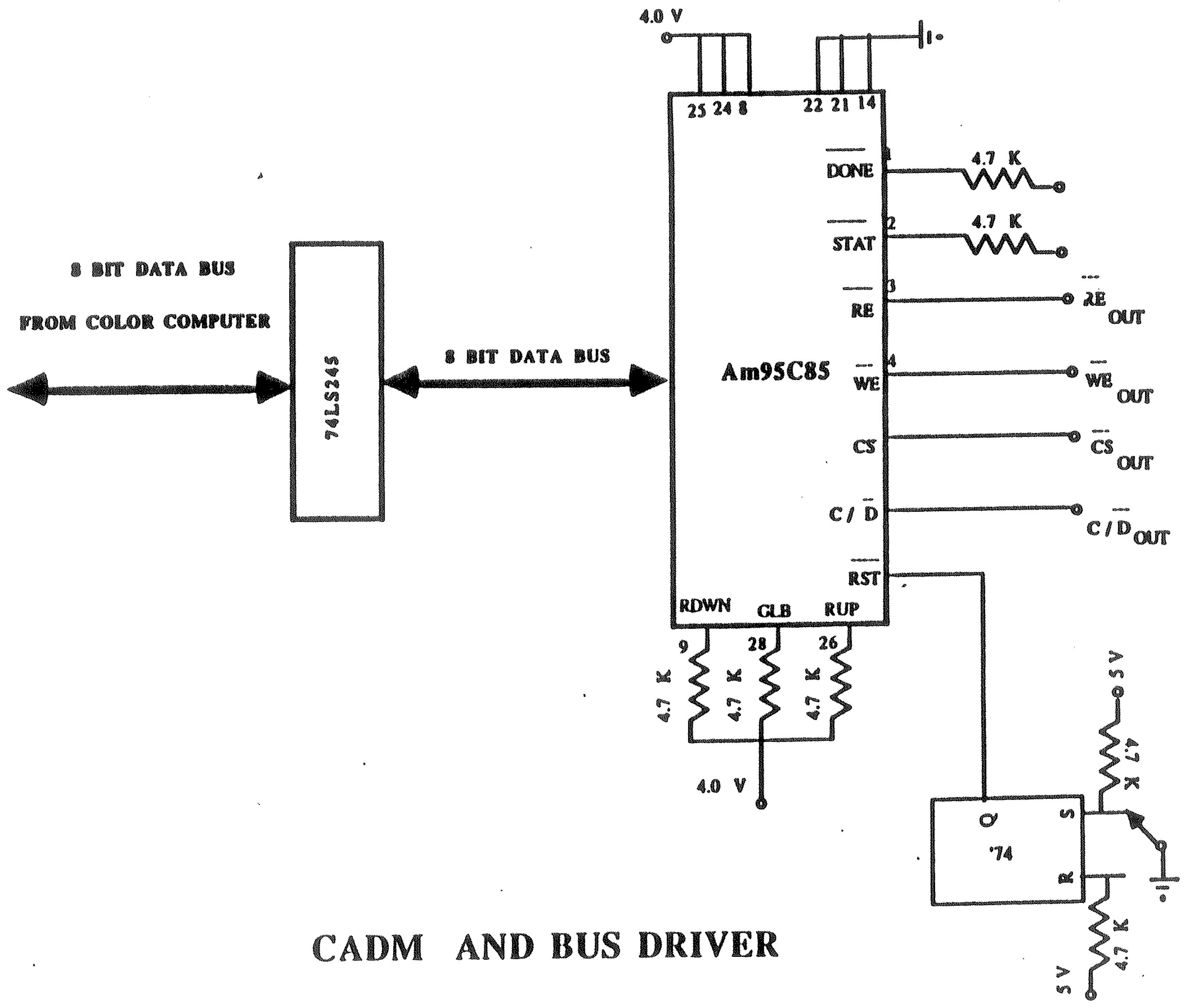


Figure 29. Block diagram of the CADM interface to TRS-80



CADM INTERFACE

Figure 30



CADM AND BUS DRIVER

Figure 31

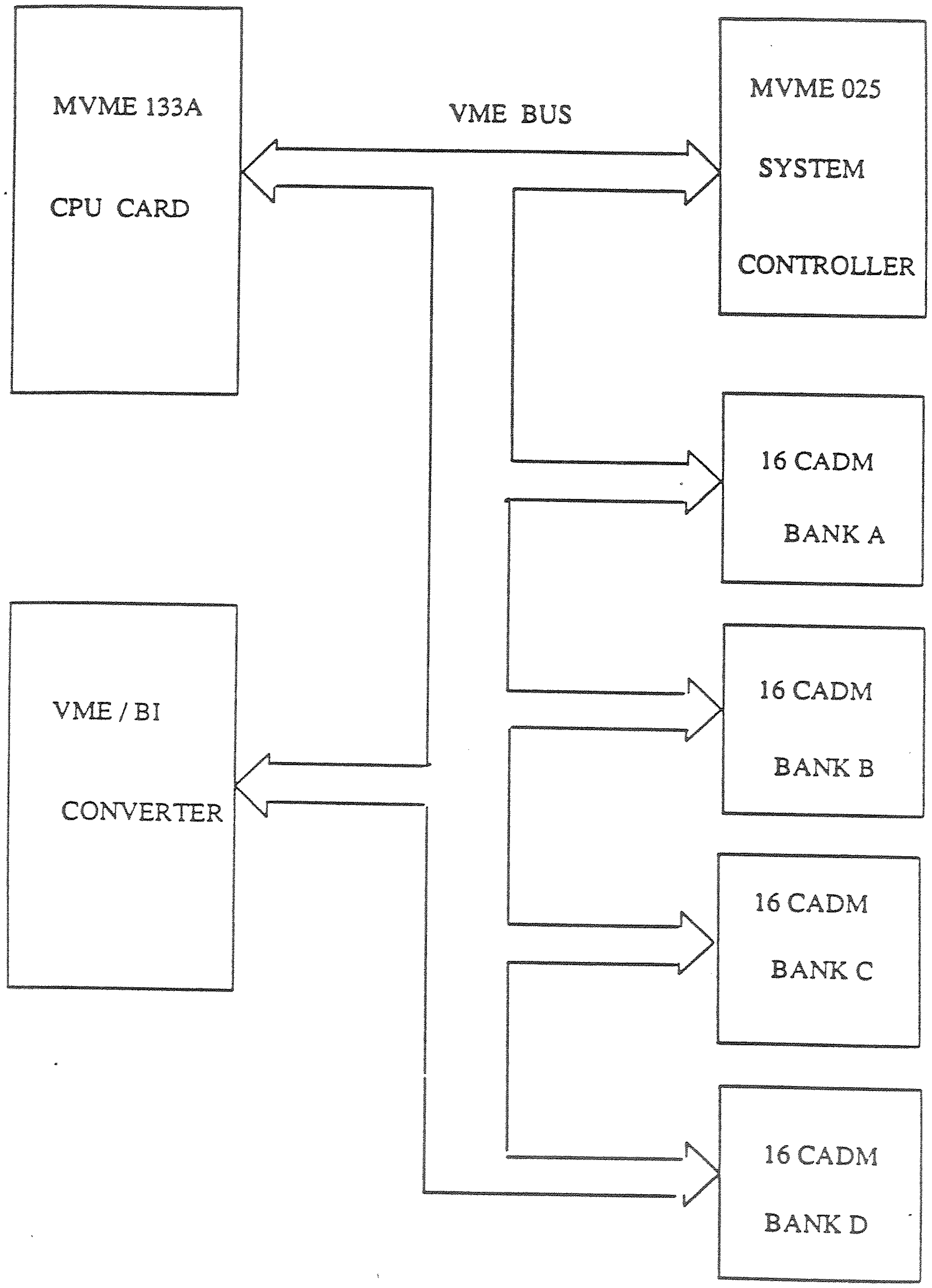


Figure 32. Block diagram of the Sort Engine with VME modules

CHAPTER 6

CONCLUSION

The design of a Sort Engine for use in database systems has been described. The Algorithms to perform Sorting, Searching, Selection and set operations such as intersection, union and difference have been presented. Performing these operations in the Sort Engine has been shown to be more efficient and at least 5 times faster than performing the same operations on a Motorola's MC68020 based SUN 3/50. The table below, gives the expected speedup for each operation.

Operation	Speedup(Min)
Sort	10
Intersection	9
Union	9
Difference	9

Table 3 : Speedup expected for small data volumes

Operation	Speedup(Min)
Sort	8
Intersection	6
Union	5
Difference	6

Table 4 : Speedup expected for large data volumes

These algorithms were also validated by simulation and comparison of results to the results obtained by performing the same operation on the SUN 3/50. The output data from simulation was compared to the output data from performing the same operations on SUN 3/50. A prototype that was built, was used to validate the timing from the simulator for all CADM commands. This comparison proved that the results of simulation are within +/- 2.5 % of actual values. These points justify the efforts in designing a special purpose system for performing the said operations. Performance of these algorithms in the sort engine are dependent on the performance of the CADM sort. As shown earlier, this is between O(N) and O(N^2). There are algorithms available that have a better order of complexity. Ultimately, this order of complexity will make the CADM inferior at large volumes of data. But its high clock speed, coupled with the design tailored for Sorting, will make it a good choice at a lower price. The performance can be further improved by designing a chip that can merge the sorted data from different arrays of CADM. These could be connected in a tree fashion to build a system that can be expanded to any size. The microengine inside the CADM can be used to design the merge chip. This will make this set of chips perform the set operations even better.

The VME module base design of the sort engine is gives higher performance than general purpose computer like SUN 3/50 at reasonable cost. Basing the design on VME modules makes it flexible and easy to upgrade and expand with new VME modules.

APPENDIX A

ALGORITHMS FOR PERFORMING OPERATIONS ON SUN 3/50

Sorting :

Quicksort [KNUT 73] is used here. The order of complexity is $N\log_2 N$, which is the best for sequential machines.

Union :

Let the lists whose union is to be found, be A and B.

```
Step1 : Sort lists A and B.

Step2 : numrecs = number of records in list B.

Step3 : i = 0.

        DONE = FALSE.

        BEGIN = Beginning of list A.

        While i < numrecs && DONE = FALSE  DO

            Get Record$_i$ from list B.

            Perform *Search* for Record$_i$ in List A.

            IF Record$_i$ not present in List A

                NEXT = Position of next higher record in list A.
            Append all records from BEGIN to NEXT to output buffer.
            Append $Record_i$ into output buffer.
            BEGIN = NEXT.
        END IF
        If END of List A,  DONE = TRUE.
    i = i+1
    End DO
```

The output buffer contains the Union of A and B in sorted order.

Intersection :

Let the lists whose intersection is to be found, be A and B.

```
**Step1** : Sort lists A and B.
**Step2** : numrecs = number of records in list B.
**Step3** : i = 0.
        DONE = FALSE.
        While i < numrecs && DONE = FALSE  DO
            Get $Record_i$ from list B.
            Perform *Search* for $Record_i$ in List A.
        IF Record$_i$ present in List A
            Append Record$_i$ to output buffer.
        END IF
        If END of List A, DONE = TRUE.
        i = i+1
    End DO
```

The output buffer now contains intersection of A and B.

Set Difference :

Let the lists whose difference is to be found, be A and B.

```
Step1 : Sort lists A and B.
Step2 : numrecs = number of records in list B.
Step3 : i = 0.
    DONE = FALSE.
    BEGIN = Beginning of list A.
    While i < numrecs && DONE = FALSE  DO
        Get Record$_i$ from list B.
        Perform *Search* for Record$_i$ in List A.
        IF Record$_i$ present in List A
            NEXT = Position of Record$_i$ in list A.
            Append all records from BEGIN to NEXT to output buffer.
            BEGIN = NEXT + Size of Record$_i$.
        END IF
        If END of List A,  DONE = TRUE.
    i = i+1
    End DO
```

Now, output buffer contains (A-B).

Selection :

The same method used in the Sort engine for *Large data volumes*, is used here.

All these routines are timed using the UNIX system call *getrusage*. To get accurate timings, these operations are repeatedly performed and the average time was taken. The simulator is written in 'C' language.

BIBLIOGRAPHY

[AMD 86] "Am95C85 (CADM) Technical Manual" *Advanced Micro Devices*, Austin, TX., 1986.

[AMD2 86] "CADM Sortmerge Performance",*Advanced Micro Devices*, Austin, TX., 1986.

[BHAT 86] Bhat, V. and Anderson, A., "CADM Based Search Utility for TRS-80 Computer", *Departmennt of Elec. Engg., Project report for EE 380N*, The University of Texas at Austin, TX., December 1986.

[BHAT 87] Bhat, V., and Jenevein, R., "Design of the CADM based Sort Engine", *Departmennt of Computer Sciences, Technical Report TR-87*, The University of Texas at Austin, TX., September 1987.

[BROW 85] Browne, J.C., A.Dale, C.Leung and R.Jenevein, "A Parallel Multi-Stage I/O Architecture with a Self-Managing Disk Cache for Data-base Management Applications", *Proceedings of the Fourth International Workshop on Database Machines*, March 1985, pp.330-345.

APPENDIX B

```
/**********************************************************************/
/*                                                                    */
/* header.h -- include file for cadm timing program                   */
/*                                                                    */
/**********************************************************************/

#define VALID
/*
#define VAXVMS
#define IBMPC
*/

#include        <stdio.h>

#define PROGNAME        "st"
#define VERSION         "1.2"

#define         MAXDATA         1000000

#define TRUE            -1
#define FALSE           0
#define NAMESIZE        80

extern  int     useMask;        /* true iff masks are used              */
extern  char    mask[];         /* contains mask bytes if useMask       */
extern  int     k;              /* number of key bytes in record        */
extern  int     kp;             /* total bytes in each record           */
extern  char    inFileName[];   /* input file name (from cmd line)      */
extern  short   sortType;       /* type of sort to perform              */
extern  char    *sortNames[];   /* names of sort types                  */
extern  int     bytesPerChip;   /* bytes of data in each chip           */
extern  char    file_a[];       /*  file containing list a               */
extern  char    file_b[];       /*  file containing list b               */


extern  long    loadClocks;     /* time to load data
*/
extern  long    sortClocks;     /* time to do sort                 */

extern  int     keyComp();      /* key comparison routine               */
extern  long    readInput();
```

```
/***********************************************************************/
/*                                                                     */
/*  This file contains the program for timing the cadm based sort engine */
/*  operations.                                                        */
/*                                                                     */
/*  Performs :                                                         */
/*                                                                     */
/*              Sorting,                                               */
/*              Selection ( >, >=, ==, <, <=),(?? !!)                  */
/*              Intersection,                                          */
/*              Union  and                                             */
/*              Set difference.                                        */
/*                                                                     */
/*  Author :  Vivekanand Bhat.                                         */
/*                                                                     */
/*  January  13, 1987                                                  */
/*                                                                     */
/***********************************************************************/


#include "header.h"

char     *d;
char     d_a[MAXDATA];          /* list a for binary operations         */
char     d_b[MAXDATA];          /* list b                               */
char     dc_a[MAXDATA];         /* copy of list a for binary operations  */
char     dc_b[MAXDATA];         /* copy of list b                       */
int      useMask;               /* true iff masks are used             */
char     mask[512];             /* contains mask bytes if useMask      */
int      k;                     /* # bytes in the key                   */
int      kp;                    /* # bytes in each record               */
FILE     *fp;
char     pred;                  /* predicate for select                */
char     filea_name[10];
char     fileb_name[10];
char     inFileName[10];
char     *counter_buff;
long     dataSize_a;            /* # bytes in list a                    */
long     dataSize_b;            /* # bytes in list b                    */
long     dataSize;
int      bytesPerChip;


main(argc, argv)
{

   struct optype
    {
       char     op_name[10];    /* name of the operation                */
       int      (*function)();  /* pointer to the function              */
       int      (*lfunction)(); /* pointer to the function for handling large data */
    }op[5];
   char *strcpy();
   char command[10];            /* command from user                   */
   extern       int     select();       /* performs selection          */
   extern       int     intersect();    /* performs intersection       */
   extern       int     unite();        /* performs union              */
   extern       int     difference();   /* performs set difference     */
   extern       int     ldifference();  /* performs set difference     */
   int          sort(); /* performs sorting           */
   extern       int     lsort2();       /* performs sorting            */
   extern       int     lselect();      /* performs selection          */
   extern       int     lintersect();   /* performs intersection       */

   long i;
   strcpy(op[0].op_name, "sort");       /* initialize ops */
   strcpy(op[1].op_name, "select");
   strcpy(op[2].op_name, "intersect");
   strcpy(op[3].op_name, "unite");
   strcpy(op[4].op_name, "diff");

   op[0].function = sort;
   op[0].lfunction = lsort2;
   op[1].function = select;
   op[1].lfunction = lselect;
   op[2].function = intersect;
   op[2].lfunction = lintersect;
   op[3].function = unite;
   op[4].function = difference;
   op[4].lfunction = ldifference;

   scancommand(argc, argv, command);    /* scan input command line */

   fp = fopen (filea_name, "r");
   if(fp == '\0')
     {
       fprintf(stderr,"\nCannot open file %s\n",filea_name);
       exit(-1);
     }
   dataSize_a = getinput( d_a, fp);
   fclose (fp);

   if(fileb_name[0] != '\0')
   {
     fp = fopen (fileb_name, "r");
     if(fp == '\0')
       {
         fprintf(stderr,"\nCannot open file %s\n",fileb_name);
         exit(-1);
       }
     dataSize_b = getinput( d_b, fp);
     fclose (fp);
   }

   for(i=0;i<5 && ((strcmp(op[i].op_name,command))!=0);i++); /* what command? */
   if(i == 5)              /* oops, wrong one */
     {
       fprintf(stderr, "\nCommand %s not found\n",command);
       exit(-1);
     }
   if(dataSize_a > 15000 || dataSize_b > 15000)
   (*op[i].lfunction)();         /* execute appropriate function                 */

   else (*op[i].function)();     /* execute appropriate function                 */

   fprintf(stderr,"\n Result written to file result.out \n");
   fprintf(stderr,"\n Statistics written to file result.stat \n");


 }


sort()
 {
   cadm(d_a,(int)dataSize_a/kp,kp,keyComp);
 }

/* getinput -- read input file and check format                            */
```

```
   long
   getinput( d, fp)
   char    *d;              /* where to put the data                  */
   FILE    *fp;             /* pointer to input file                  */
   {
   int     c;
   long    dataSize;

    dataSize = 0;
    while ( (c=fgetc(fp)) != EOF )
    {
      d[ dataSize++] = c;
      if ( dataSize >= MAXDATA ) {
      fprintf( stdout, "too much data--change MAXDATA\n");
      exit( -1);
      }
      if ( c == 0 ) {
      fprintf( stdout, "%s%s%s",
      "The c language is picky about binary zero's.\n",
      "Please remove them from your input file.\n",
      "(The CADM has no problems with them.)\n");
      exit( -1);
      }
    }
    if ( dataSize % kp != 0 && (dataSize % (k+1))!=0) {
        fprintf( stdout, "input is not integral number of records\n");
        exit( -1);
        }

        return( dataSize);
}
```

```
/************************************************************************/
/*                                                                      */
/* scancommand.c -- command line scanner for sort engine routines      */
/*                                                                      */
/************************************************************************/

#include        "header.h"
char    filea_name[];   /*  file containing list a                      */
char    fileb_name[];   /*  file containing list b                      */

scancommand( argc, argv, command)
        int     argc;
        char    *argv[], *command;
{
        int     i;
        int     j;
        int     arg;
        int     p;
        int     la;
        char    *strcpy();
        extern  char    pred;

        k = -1;
        p = -1;
        la = -1;
        useMask = FALSE;
        filea_name[0] = '\0';
        fileb_name[0] = '\0';
        if ( argc == 1 ) {
                fprintf( stdout, "%s%s%s%s%s%s%s%s%s%s%s",
                        "software OR cadm", " -- calculate timing sort engine\n",
                        "software OR cadm", " operation -k #keybytes -p #pointerBytes [-l
la]",
                        " -P predicate",
                        " [-m 0HH ...]  file_a file_b\n",
                        "    0HH is hex mask byte.\n",
                        " operation :  sort, select,intersection,union and difference.",
                        "predicate : '==', '>=', '>', '<', '<=', '<>' (range).\n",
                        "    0HH is hex mask byte.\n",
                        "    la defaults to max allowed (cadm only)\n"

                        );
                exit( -1);
                }
        if(strlen(argv[1]) > 9)
         {
           fprintf(stderr,"Command %s is too long\n",argv[1]);
           exit(-1);
         }
        else strcpy(command,argv[1]);

        for( arg = 2; arg < argc; arg++) {
                if ( argv[arg][0] == '-' ) {
                        switch (argv[ arg][ 1]) {
                        case 'k':
                                arg++;  /* skip -k */
                                if ( arg >= argc ) {
                                        printf("-k requires number\n");
                                        exit(-1);
                                        }
                                k = atoi( argv[arg] );
                                break;
                        case 'p':
                                arg++;  /* skip -p */
                                if ( arg >= argc ) {
                                        printf("-p requires number\n");
                                exit(-1);
                                }
                        p = atoi( argv[arg] );
                        break;
                case 'l':
                        arg++;  /* skip -l */
                        if ( arg >= argc ) {
                                printf("-l requires number\n");
                                exit(-1);
                        }
                        la = atoi( argv[arg] );
                        break;

                case 'm':
                        arg++;  /* skip -m */
                        useMask = TRUE;
                        fprintf( stdout, "mask bytes:");
                        i = 0;
                        while ( arg < argc && argv[arg][0] == '0' ) {
                                sscanf( argv[arg], "%x", &j);
                                mask[i++] = (char) j;
                                fprintf( stdout, "  %x", j);
                                arg++;
                                }
                        if ( arg != argc ) arg--;
                        fprintf( stdout, "\n");
                        break;
                case 'P':
                        arg++;
                        if ( arg >= argc ) {
                                printf("-P requires string\n");
                                exit(-1);
                                }
                        if(strcmp("==",argv[arg]) == 0)pred = 'a';
                        else if(strcmp(">",argv[arg]) == 0)pred = 'b';
                        else if(strcmp(">=",argv[arg]) == 0)pred = 'c';
                        else if(strcmp("<",argv[arg]) == 0)pred = 'd';
                        else if(strcmp("<=",argv[arg]) == 0)pred = 'e';
                        else if(strcmp("<>",argv[arg]) == 0)pred = 'f';
                        else
                          {
                            fprintf(stderr,"\n Predicate %s unknown\n",
                            argv[arg]);
                            exit(-1);
                          }
                        break;

                default:
                        printf( "bad option %s\n", argv[arg]);
                        break;
                        };  /* end switch */
                }  /* end if '-' */
        else {  /* not an option */
                /* must be a file name */
                if ( strlen( argv[arg]) >= 10 ) {
                        fprintf( stdout, "file name '%s' too long\n",
                                argv[arg]);
                        exit( -1);
                        }
                else if ( filea_name[0] == '\0' ) {
                        strcpy( filea_name, argv[arg]);
                        }
                else if ( fileb_name[0] == '\0' ) {
                        strcpy( fileb_name, argv[arg]);
                        }
                else {
```

```
/************************************************************************/
/*                                                                      */
/* driver for cadm sort merge program and timing routines               */
/* sortmerge algorithm taken straight out of Knuth's Bible              */
/*                                                                      */
/************************************************************************/

#include        "header.h"
#ifdef  VALID
#include        <sys/time.h>
#include        <sys/resource.h>
#endif
#ifdef  IBMPC
#include        <dos.h>
#endif

        char    d[MAXDATA];     /* data to sort                         */
        char    dc[MAXDATA];    /* copy of data to sort with a tag      */
        int     useMask;        /* true iff masks are used              */
        char    mask[512];      /* contains mask bytes if useMask       */
        int     k;              /* number of key bytes in record        */
        int     c;              /* number of cadm chips used            */
        int     kp;             /* total bytes in each record           */
        int     kpact;          /* total bytes in each record           */
        int     bytesPerChip;   /* bytes of data in each chip           */
        char    inFileName[NAMESIZE];   /* input file name (cmd line)   */
        long    milliseconds;
        long    dataSize;       /* cadm buffer size                     */

        struct bin_info_type
        {
                int     size;           /* Size of bin in recs          */
                char    *top;           /* current top of the bin       */
        }bin_info[30];
#ifdef  VALID
struct  rusage          startTime;
struct  rusage          endTime;
#endif  VALID
#ifdef  VAXVMS
struct  tbuffer {
                int     proc_user_time;
                int     proc_system_time;
                int     child_user_time;
                int     child_system_time;
                };
struct  tbuffer startTime;
struct  tbuffer endTime;
#endif  VAXVMS
#ifdef  IBMPC
struct  reg             startTime;
struct  reg             endTime;
#endif

main(argc, argv)
        int     argc;
        char    *argv[];
{
        long    inputsize;      /* actual size of input                 */
        float   time;

        long    copyCount;
        long    sortCount;
        long    copyTime;
        long    counter;
        long    sysCallTime;    /* time for a system call to check time */
        long    temp1,temp2;
        register int i=0,j=0,k=0,l=0;   /* loop counters */
        register int    numbins;        /* # bins for sort merge                    */
        char    *ptr,*from,*to,*calloc();
        char    *counter_buff;  /* image of actual sequence of tags */
        FILE    *fpres,*fpout,*fopen();


        scanCmd( argc, argv);
        inputsize = readInput(d);
        dataSize = (bytesPerChip * c);

        /* take care of the odd man out, if any */

        j = inputsize % dataSize;
        numbins = inputsize / dataSize + (j>0);
        k = numbins - (j>0);
        if(j)
        {
                bin_info[k].top = &d[inputsize - j];
                bin_info[k].size = j / kp;
                cadm(bin_info[k].top,bin_info[k].size,kp,keyComp);
        }

        /* setup bins and also sort them */

        l = kp;
        j = 0;
        for(i=0;i<k;i++)
        {
                bin_info[i].top = &(d[j]);
                cadm(&(d[j]),dataSize / l,l,keyComp);
                bin_info[i].size = dataSize / l;
                j += dataSize;
        }

        for(counter=0;counter<inputsize;counter++)
                dc[counter]=d[counter];
sortCount=0;

/* all set for merging */
#ifdef  VALID
        getrusage( RUSAGE_SELF, &startTime);
        endTime.ru_utime = startTime.ru_utime;
        while( endTime.ru_utime.tv_sec - startTime.ru_utime.tv_sec < 30 ) {
#endif  VALID

        sortCount++;
        /* take care of the odd man out, if any */

        j = inputsize % dataSize;
        numbins = inputsize / dataSize + (j>0);
        k = numbins - (j>0);
        if(j)
        {
                bin_info[k].top = &d[inputsize - j];
                bin_info[k].size = j / kp;
        }

        /* setup bins and also sort them */

        l = kp;
        j = 0;
        for(i=0;i<k;i++)
        {
                bin_info[i].top = &(d[j]);
                bin_info[i].size = dataSize / l;
```

```
                                        fprintf( stdout, "too many filenames\n");
                                        }
                                }
                        };  /* end for */
                if ( k == -1 ) {
                        fprintf( stdout, "value of k defaults to 8\n");
                        k = 8;
                        }
                if ( p == -1 ) {
                        fprintf( stdout, "value of p defaults to 2\n");
                        p = 2;
                        }
                kp = k + p;

                if ( la == -1 ) {
                        if ( useMask ) {
                                la = (1024 - k - kp) / kp;  /* # records */
                        }
                        else {
                                la = (1024 - kp) / kp;  /* # records */
                        }
                        bytesPerChip = la * kp;
                }
                else {
                bytesPerChip = la + 1 - ( useMask ? k : 0);
                if ( (bytesPerChip % kp) != 0 ) {
                fprintf( stdout, "bad la value.\n");
                }
                }

        }  /* end scan cmd line */
```

```
/*********************************************************************/
/*                                                                   */
/*  intersect.c     : finds  the intersection of two lists and returns  */
/*  the time required.                                               */
/*                                                                   */
/*                                                                   */
/* April 2, 1987                                                     */
/*                                                                   */
/*********************************************************************/


#include        "header.h"
#define MICROSEC        0.0625  /* 1/(16MHz) in m-sec                */

extern  long    fndTime();
extern  long    fndInitTime();


extern  int     bytesPerChip;
extern  int     useMask;                /* true iff masks are used       */
extern  char    mask[512];              /* contains mask bytes if useMask*/
extern  char    *d;
extern  char    d_a[MAXDATA];           /* list a for binary operations  */
extern  char    d_b[MAXDATA];           /* list b                        */
extern  char    dc_a[MAXDATA];          /* copy of list a for binary operations
*/
extern  char    dc_b[MAXDATA];          /* copy of list b                */
extern  int     k;                      /* # bytes in the key            */
extern  int     kp;                     /* # bytes in each record        */
extern  char    filea_name[10];
extern  char    fileb_name[10];
extern  long    dataSize_a;             /* # bytes in list a             */
extern  long    dataSize_b;             /* # bytes in list b             */
extern  long    dataSize;


extern  long    loadClocks;
extern  long    broadcastClocks;
extern  long    searchClocks;
extern  long    pushClocks;
extern  long    sortClocks;
char    *addrptr;

intersect()
{
        char    *skey;
        long    i;              /* for copying data */
        int     status;         /* status pin of the cadm */
        int     done;           /* done pin of the cadm */
        int     numrecs;        /* # records in the smaller list in cadm */
        FILE    *fp,*fpout,*fpstat;
        long    load_a;
        long    sort_a;
        long    load_b;
        long    sort_b;
        char    *ptr;
        char    *result;        /* result of intersection */
        long    total;          /* total clockcycles required */
        char    *malloc();
        char    *list;


/*  sort each input list  */

        fclose(stdout);
        cadm(d_a,(int)dataSize_a/kp,kp,keyComp);
        load_a = loadClocks;
        sort_a = sortClocks;
        loadClocks = 0;
        broadcastClocks = 0;
        searchClocks = 0;
        pushClocks = 0;
        sortClocks = 0;
        cadm(d_b,(int)dataSize_b/kp,kp,keyComp);
        if(sortClocks < sort_a) sortClocks = sort_a;


        if(dataSize_b / kp > dataSize_a / kp)
          {
             d = d_b;
             list  = d_a;
             dataSize = dataSize_b;
             numrecs = dataSize_a / kp;
          }
        else
          {
             d = d_a;
             list = d_b;
             dataSize = dataSize_a;
             numrecs = dataSize_b / kp;
          }


        loadClocks += load_a;   /* loading in the first list */

/* now, d points to larger list in cadm, dataSize is its size,  */
/*list points to smaller list in other cadm array  with numrecs recs.*/

        total = 0;
        skey = malloc(k);

        ptr = list;
        result = list;
        done = 0;


        while(numrecs && !done)
        {
          numrecs--;
          strncpy(skey,ptr,k);
          total += fndTime(skey, dataSize);
          total += 8 * k;   /* for fetching key from one cadm list */
          addrptr = d + binarySearch2(skey,d,dataSize);
          status = keyComp(skey, addrptr);
          done = (addrptr >= (d + dataSize));
          if(!status)
            {
              for(i=0;i<kp;i++) *result++ = *ptr++;
              total += 8 * kp;  /* for reading record back to answer buffer */
            }
          else ptr += kp;
        }


fp = fopen("result.out","w");
ptr = list;
for(i=0;i<result - list;i++) fprintf(fp,"%c",*ptr++);
fpstat = fopen("result.stat","w");      /* statistics */

fprintf(fpstat,"\n\n\tCADM  SORT ENGINE  SIMULATOR  STATISTICS\n\n");
fprintf(fpstat,"\t\tOperation :\t\tIntersection\n");
fprintf(fpstat,"\n\t\tFile_A :\t\t\t%s\n",filea_name);
fprintf(fpstat,"\n\t\tFile_B :\t\t\t%s\n",fileb_name);
fprintf(fpstat,"\n\t\t# Records_A :\t\t%d\n",dataSize_a / kp);
fprintf(fpstat,"\n\t\t# Records_B :\t\t%d\n",dataSize_b / kp);
fprintf(fpstat,"\n\t\tKey length : \t\t%d\n",k);
fprintf(fpstat,"\n\t\tPointer length : \t%d\n",kp-k);
fprintf(fpstat, "\n\t\t# Sort Clocks : \t%7ld\n", loadClocks + sortClocks);
fprintf(fpstat, "\n\t\t# Clock cycles : \t%7ld\n", total);
fprintf(fpstat,"\n\t\tSort time : \t\t%8.3f millisecs\n",((loadClocks + sortClocks) * MICR
OSEC) / 1000);
total += loadClocks + sortClocks;
fprintf(fpstat,"\n\t\tTotal time : \t\t%8.3f millisecs\n",(total * MICROSEC) / 1000);
fprintf(fpstat,"\n\n\t\t\t*********************\n");
fclose(fp);
fclose(fpstat);

}  /* end of main */
```

```
/*********************************************************************/
/*                                                                   */
/* select.c -- calculates the time necessary to find all occurences  */
/* of a given record.  i.e., returns time required to perform selection*/
/* on a list in cadm.                                                */
/*                                                                   */
/* April 9, 1987                                                     */
/*                                                                   */
/*********************************************************************/

#include        "header.h"
#define MICROSEC        0.0625  /* 1/(16MHz) in m-sec               */

extern  long    fndTime();
extern  long    fndInitTime();


extern  int     bytesPerChip;
extern  int     useMask;                /* true iff masks are used      */
extern  char    mask[512];              /* contains mask bytes if useMask*/
extern  char    *d;
extern  char    d_a[MAXDATA];           /* list a for binary operations  */
extern  char    d_b[MAXDATA];           /* list b                        */
extern  char    dc_a[MAXDATA];          /* copy of list a for binary operations */
extern  char    dc_b[MAXDATA];          /* copy of list b                */
extern  int     k;                      /* # bytes in the key            */
extern  int     kp;                     /* # bytes in each record        */
extern  char    filea_name[10];
extern  char    fileb_name[10];
extern  long    dataSize_a;             /* # bytes in list a             */
extern  long    dataSize_b;             /* # bytes in list b             */
extern  long    dataSize;
char    *addrptr;
extern  char    pred;           /* selection criterion :
                                        =  ==  a
                                        >  ==  b
                                        >= ==  c
                                        <  ==  d
                                        <= ==  e
                                     range     ==  f   */


select()
{
        char    *skey;
        char    inLine[256];
        long    total;
        int     count;
        long    thisTime;
        long    maxTime;
        long    minTime;
        long    i;              /* for copying data */
        int     status;         /* status pin of the cadm */
        int     done;           /* done pin of the cadm */
        int     stop;
        float   wallTime;
        FILE    *fpout, *fpstat;
        extern  long    totalClocks;

        d = d_a;
        dataSize = dataSize_a;
        fclose(stdout);
        cadm(d,(int)dataSize/kp,kp,keyComp);

        /* reopen the to-find file */
        if ( fileb_name[0] == '\0' ) {
```

```
        freopen( "/dev/tty", "r", stdin);
        }
else {
        if (freopen( fileb_name, "r", stdin) == NULL) {
                fprintf( stdout, "cannot open %s\n", fileb_name);
                exit( 1);
                }
        }

fpout = fopen("result.out","w");
total = 0;
while( gets( inLine)) {
  if ( strlen(inLine) != k )
  printf("Your key \"%s\" is not %d bytes long.\n", inLine,k);
  switch(pred)
   {
     case 'a' :          /*  =   */
        thisTime = fndTime( inLine, dataSize);
        total += thisTime;
        addrptr =  d + binarySearch2(inLine,d,dataSize);
        status = keyComp( inLine, addrptr);
        stop = status;
        while (! stop)
        {
          for (i=0;i<kp;i++)fputc(*addrptr++,fpout);
          thisTime = 14 + keyCompare( inLine, addrptr, &i);
          total += thisTime;
          stop = keyComp( inLine, addrptr);
        }
        break;

    case 'b' :           /*  >  */
        thisTime = fndTime( inLine, dataSize);
        total += thisTime;
        addrptr =  d + binarySearch2(inLine,d,dataSize);
        status = keyComp( inLine, addrptr);
        stop = status;
        while (! stop)    .
        {
          addrptr += kp;
          thisTime = 14 + keyCompare( inLine, addrptr, &i);
          total += thisTime;
          stop = keyComp( inLine, addrptr);
        }
        i = ( d + dataSize - addrptr );
        total += i * 6;                  /* for reading back */
        while(i--) fputc(*addrptr++,fpout);
        break;

    case 'c' :           /*  >=  */
        thisTime = fndTime( inLine, dataSize);
        total += thisTime;
        addrptr =  d + binarySearch2(inLine,d,dataSize);
        status = keyComp( inLine, addrptr);
        i = ( d + dataSize - addrptr );
        total += i * 6;                  /* for reading back */
        while(i--) fputc(*addrptr++,fpout);
        break;
   }


}
total += totalClocks;
fpstat = fopen("result.stat","w");       /* statistics */
```

```
        fprintf(fpstat,"\n\n\tCADM  SORT ENGINE  SIMULATOR  STATISTICS\n\n");
        fprintf(fpstat,"\t\tOperation :\t\tSelection\n");
        fprintf(fpstat,"\n\t\tFile_A :\t\t\t%s\n",filea_name);
        fprintf(fpstat,"\n\t\tFile_B :\t\t\t%s\n",fileb_name);
        fprintf(fpstat,"\n\t\t# Records_A :\t\t%d\n",dataSize_a / kp);
        fprintf(fpstat,"\n\t\t# Records_B :\t\t%d\n",dataSize_b / kp);
        fprintf(fpstat,"\n\t\tKey length : \t\t%d\n",k);
        fprintf(fpstat,"\n\t\tPointer length : \t%d\n",kp-k);
        fprintf(fpstat, "\n\t\t# Clock cycles : \t%7ld\n", total);
        fprintf(fpstat,"\n\t\tTotal time : \t\t%8.3f millisecs\n",(total * MICROSEC) / 100
0);
        fprintf(fpstat,"\n\n\t\t\t*********************\n");
        fclose(fpout);
        fclose(fpstat);

        }  /* end of main */
```

```
/***********************************************************************/
/*                                                                     */
/*  union.c  : finds  the intersection of two lists and returns        */
/*  the time required.                                                 */
/*                                                                     */
/*                                                                     */
/* April 2, 1987                                                       */
/*                                                                     */
/***********************************************************************/


#include        "header.h"
#define MICROSEC        0.0625  /* 1/(16MHz) in m-sec                  */



extern  long    fndTime();
extern  long    fndInitTime();


extern  int     bytesPerChip;
extern  int     useMask;                /* true iff masks are used       */
extern  char    mask[512];              /* contains mask bytes if useMask*/
extern  char    *d;
extern  char    d_a[MAXDATA];           /* list a for binary operations  */
extern  char    d_b[MAXDATA];           /* list b                        */
extern  char    dc_a[MAXDATA];          /* copy of list a for binary operations
*/
extern  char    dc_b[MAXDATA];          /* copy of list b                */
extern  int     k;                      /* # bytes in the key            */
extern  int     kp;                     /* # bytes in each record        */
extern  char    filea_name[10];
extern  char    fileb_name[10];
extern  long    dataSize_a;             /* # bytes in list a             */
extern  long    dataSize_b;             /* # bytes in list b             */
extern  long    dataSize;

extern  long    loadClocks;
extern  long    broadcastClocks;
extern  long    searchClocks;
extern  long    pushClocks;
extern  long    sortClocks;
char    *addrptr;

unite()
{
        char    *skey;
        long    i;              /* for copying data */
        int     status;         /* status pin of the cadm */
        int     done;           /* done pin of the cadm */
        int     numrecs;        /* # records in the smaller list in cadm */
        FILE    *fp,*fpout,*fpstat;
        long    load_a;
        long    sort_a;
        long    load_b;
        long    sort_b;
        char    *ptr;
        char    *result;        /* result of union */
        long    total;          /* total clockcycles required */
        char    *malloc();
        char    *list;


/*  sort each input list  */
```

```
        fclose(stdout);
        cadm(d_a,(int)dataSize_a/kp,kp,keyComp);
        load_a = loadClocks;
        sort_a = sortClocks;
        loadClocks = 0;
        broadcastClocks = 0;
        searchClocks = 0;
        pushClocks = 0;
        sortClocks = 0;
        cadm(d_b,(int)dataSize_b/kp,kp,keyComp);
        if(sortClocks < sort_a) sortClocks = sort_a;


        if(dataSize_b / kp > dataSize_a / kp)
          {
             d = d_b;
             list  = d_a;
             dataSize = dataSize_b;
             numrecs = dataSize_a / kp;
          }
        else
          {
             d = d_a;
             list = d_b;
             dataSize = dataSize_a;
             numrecs = dataSize_b / kp;
          }


        loadClocks += load_a;   /* loading in the first list */

/* now, d points to larger list in cadm, dataSize is its size,   */
/*list points to smaller list in cadm with numrecs recs.         */

        total = 0;
        skey = malloc(k);

        ptr = list;
        result = list;
        done = 0;


        while(numrecs)
        {
          numrecs--;
          strncpy(skey,ptr,k);
          total += 8 * kp;      /* read record from smaller list */
          total += fndTime(skey, dataSize);
          addrptr = d + binarySearch2(skey,d,dataSize);
          status = keyComp(skey, addrptr);
          if(status)  /* if record does not exist in large list */
           {
             strncpy(skey,ptr,kp);
             total += push(skey, addrptr, d, dataSize);  /* push record in place */
             dataSize += kp;
           }

          ptr += kp;
        }

/* now the cadm array a contains the answer */

total += dataSize * 6;  /* for loading answer from cadm */
```

```
fp = fopen("result.out","w");
d[dataSize] = '\0';
fprintf(fp,"%s",d);

fpstat = fopen("result.stat","w");      /* statistics */

fprintf(fpstat,"\n\n\tCADM  SORT ENGINE  SIMULATOR  STATISTICS\n\n");
fprintf(fpstat,"\t\tOperation :\t\tUnion\n");
fprintf(fpstat,"\n\t\tFile_A :\t\t\t%s\n",filea_name);
fprintf(fpstat,"\n\t\tFile_B :\t\t\t%s\n",fileb_name);
fprintf(fpstat,"\n\t\t# Records_A :\t\t%d\n",dataSize_a / kp);
fprintf(fpstat,"\n\t\t# Records_B :\t\t%d\n",dataSize_b / kp);
fprintf(fpstat,"\n\t\tKey length : \t\t%d\n",k);
fprintf(fpstat,"\n\t\tPointer length : \t%d\n",kp-k);
fprintf(fpstat, "\n\t\t# Sort Clocks : \t%7ld\n", loadClocks + sortClocks);
fprintf(fpstat, "\n\t\t# Clock cycles : \t%7ld\n", total);
fprintf(fpstat,"\n\t\tSort time : \t\t%8.3f millisecs\n",((loadClocks + sortClocks) * MICR
OSEC) / 1000);
total += loadClocks + sortClocks;
fprintf(fpstat,"\n\t\tTotal time : \t\t%8.3f millisecs\n",(total * MICROSEC) / 1000);
fprintf(fpstat,"\n\n\t\t\t*********************\n");
fclose(fpout);
fclose(fp);


}  /* end of main */




push(skey, addrptr, start, dataSize)
 char   *skey;          /* record to  push */
 char   *addrptr;       /* place to put the record */
 char   *start;         /* where the array starts */
 long   dataSize;       /* current total size of the array */
{
  int   i;
  extern        long    pushTime();

  shift(addrptr, addrptr+kp, ((start + dataSize) - addrptr)); /* move bytes */
  for(i=0;i<kp;i++)
   *addrptr++ = skey[i];        /* insert new record */
  return((int)pushTime(dataSize / bytesPerChip));
}
```

```
/*********************************************************************/
/*                                                                   */
/*  diff.c  : finds  the difference of two lists and returns         */
/*  the time required.                                               */
/*                                                                   */
/*                                                                   */
/* April 2, 1987                                                     */
/*                                                                   */
/*********************************************************************/


#include        "header.h"
#define MICROSEC        0.0625  /* 1/(16MHz) in m-sec                */

extern  long    fndTime();
extern  long    fndInitTime();


extern  int     bytesPerChip;
extern  int     useMask;                /* true iff masks are used       */
extern  char    mask[512];              /* contains mask bytes if useMask*/
extern  char    *d;
extern  char    d_a[MAXDATA];           /* list a for binary operations  */
extern  char    d_b[MAXDATA];           /* list b                        */
extern  char    dc_a[MAXDATA];          /* copy of list a for binary operations
*/
extern  char    dc_b[MAXDATA];          /* copy of list b                */
extern  int     k;                      /* # bytes in the key            */
extern  int     kp;                     /* # bytes in each record        */
extern  char    filea_name[10];
extern  char    fileb_name[10];
extern  long    dataSize_a;             /* # bytes in list a             */
extern  long    dataSize_b;             /* # bytes in list b             */
extern  long    dataSize;



extern  long    loadClocks;
extern  long    broadcastClocks;
extern  long    searchClocks;
extern  long    pushClocks;
extern  long    sortClocks;

difference()
{
        char    *skey, *addrptr;
        long    i;              /* for copying data */
        int     status;         /* status pin of the cadm */
        int     done;           /* done pin of the cadm */
        int     numrecs;        /* # records in the smaller list in cadm */
        FILE    *fp, *fpstat, *fpout;
        long    load_a;
        long    sort_a;
        long    load_b;
        long    sort_b;
        char    *ptr;
        long    total;          /* total clockcycles required */
        char    *malloc();
        char    *list;


/*  sort each input list  */

        fclose(stdout);
        cadm(d_a,(int)dataSize_a/kp,kp,keyComp);
        load_a = loadClocks;
        sort_a = sortClocks;
        loadClocks = 0;
        broadcastClocks = 0;
        searchClocks = 0;
        pushClocks = 0;
        sortClocks = 0;
        cadm(d_b,(int)dataSize_b/kp,kp,keyComp);
        if(sortClocks < sort_a) sortClocks = sort_a;

        d = d_a;
        dataSize = dataSize_a;
        numrecs = dataSize_b / kp;
        loadClocks += load_a;   /* loading in the first list */


        total = 0;
        skey = malloc(k);

        ptr = d_b;
        done = 0;

        while(numrecs && !done)
        {
          numrecs--;
          strncpy(skey,ptr,k);
          total += 8 * k;        /* read key from list a */
          total += fndTime(skey, dataSize);
          addrptr = d + binarySearch2(skey,d,dataSize);
          status = keyComp(skey, addrptr);
          done = (addrptr >= (d + dataSize));
          if(! status)  /* if record does exist in large list */
           {
             total += pop(addrptr, d, dataSize);  /* pop record out */
             dataSize -= kp;
           }

          ptr += kp;
        }

/* now the cadm array a contains the answer */

total += dataSize * 6;  /* for loading answer from cadm */



fp = fopen("result.out","w");
d[dataSize] = '\0';
fprintf(fp,"%s",d);


fpstat = fopen("result.stat","w");      /* statistics */

fprintf(fpstat,"\n\n\tCADM  SORT ENGINE  SIMULATOR  STATISTICS\n\n");
fprintf(fpstat,"\t\tOperation :\t\tDifference\n");
fprintf(fpstat,"\n\t\tFile_A :\t\t\t%s\n",filea_name);
fprintf(fpstat,"\n\t\tFile_B :\t\t\t%s\n",fileb_name);
fprintf(fpstat,"\n\t\t# Records_A :\t\t%d\n",dataSize_a / kp);
fprintf(fpstat,"\n\t\t# Records_B :\t\t%d\n",dataSize_b / kp);
fprintf(fpstat,"\n\t\tKey length : \t\t%d\n",k);
fprintf(fpstat,"\n\t\tPointer length : \t%d\n",kp-k);
fprintf(fpstat, "\n\t\t# Sort Clocks : \t%7ld\n", loadClocks + sortClocks);
fprintf(fpstat, "\n\t\t# Clock cycles : \t%7ld\n", total);
fprintf(fpstat,"\n\t\tSort time : \t\t%8.3f millisecs\n",((loadClocks + sortClocks) * MICR
OSEC) / 1000);
total += loadClocks + sortClocks;
fprintf(fpstat,"\n\t\tTotal time : \t\t%8.3f millisecs\n",(total * MICROSEC) / 1000);
fprintf(fpstat,"\n\n\t\t\t*********************\n");
fclose(fpout);
fclose(fpstat);

}  /* end of main */



pop(addrptr, start, dataSize)
 char   *addrptr;       /* place to take record from*/
 char   *start;         /* where the array starts */
 long   dataSize;       /* current total size of the array */
{
  extern        long    pushTime();
  long  howmany;        /* # bytes to be moved */
  char  *from;
  char  *to;

  from = addrptr + kp;
  to = addrptr;
  howmany = (start + dataSize) - addrptr;

  while(howmany--)
    *to++ = *from++;
  return((int)pushTime(dataSize / bytesPerChip));
 }
```

```
/************************************************************************/
/*                                                                      */
/* shift.c -- shifts a block of bytes to higher memory location        */
/*                                                                      */
/************************************************************************/

#define reg     register

int
shift( from, to, howMany)
reg     char    *from;          /* start of bytes to move               */
reg     char    *to;            /* where to put them                    */
reg     long    howMany;        /* number of bytes to move              */
{
        from += howMany;
        to += howMany;
        while( howMany-- ) {
                *(--to) = *(--from);
                }

        }  /* end shift */
```

```
                  j += dataSize;
          }

          for(counter=0;counter<inputsize;counter++)
                  d[counter]=dc[counter];

          j = 1;
          k = 2;
          while(1)
          {
                  for(i = 0; i< numbins; i += k)
                  {
                          l = i + j;
                          if(l < numbins) merge(i,l);     /* merge bins i and j , put the re
sult back starting at top of bin i. */
                  }
                  j *= 2;
                  k *= 2;
                  if(j > numbins) break;
          }
#ifdef VALID
                  getrusage(RUSAGE_SELF,&endTime);
#endif VALID
}

#ifdef  VALID
        temp1 = 1000000 *
          ( endTime.ru_utime.tv_sec - startTime.ru_utime.tv_sec ) +
          ( endTime.ru_utime.tv_usec - startTime.ru_utime.tv_usec );
        temp1 += 1000000 *      /* system */
          ( endTime.ru_stime.tv_sec - startTime.ru_stime.tv_sec ) +
          ( endTime.ru_stime.tv_usec - startTime.ru_stime.tv_usec );
        temp1 = temp1 / sortCount;
#endif  VALID
        fpout = fopen("sortmerge.out","w");
        d[inputsize] = '\0';
        fprintf(fpout,"%s",d);
        fclose(fpout);

#ifdef  VALID
        /* time copy on a VALID */
        getrusage( RUSAGE_SELF, &startTime);
        endTime.ru_utime = startTime.ru_utime;
        while( endTime.ru_utime.tv_sec - startTime.ru_utime.tv_sec < 30 ) {
        for(counter=0;counter<inputsize;counter++)
                d[counter]=dc[counter];
                copyCount++;
                getrusage( RUSAGE_SELF, &endTime);
                }
        copyTime = 1000000 *
                ( endTime.ru_utime.tv_sec - startTime.ru_utime.tv_sec ) +
                ( endTime.ru_utime.tv_usec - startTime.ru_utime.tv_usec );
        copyTime += 1000000 *   /* system */
                ( endTime.ru_stime.tv_sec - startTime.ru_stime.tv_sec ) +
                ( endTime.ru_stime.tv_usec - startTime.ru_stime.tv_usec );
#endif  VALID

        temp2 = copyTime/copyCount;
        milliseconds = (temp1-temp2)/1000;
        softwr_time = (float)milliseconds;

        fpres = fopen("sortmerge.res","a");
        fprintf(fpres,"Input File:  %s \n\n",inFileName);
        fprintf(fpres,"# CADM Chips:    %d\n\n",c);
        fprintf(fpres,"Offline Sort:    %8.3f milliseconds\n",offlnsrt_time);
        fprintf(fpres,"Software Time:   %8.3f milliseconds\n",softwr_time);
        fprintf(fpres,"Total Time:      %8.3f milliseconds\n\n",(softwr_time + offlnsrt_ti
me ));
        fclose(fpres);
}
```

```
/**********************************************************************/
/*  two way merge. algorithm is right out of Knuth's Bible.          */
/**********************************************************************/

#include "header.h"

merge(bin1,bin2)
int     bin1,bin2;      /* index of bins to be merged */
{

        extern struct bin_info_type
        {
                int     size;           /* Size of bin in bytes        */
                char    *top;           /* current top of the bin      */
        }bin_info[30];
        char    dc[MAXDATA];    /*  buffer for output of merge */
        register        char    *top1;  /* top of first set */
        register        char    *top2;  /* top of second set */
        register        char    *topout = dc;/* top of output */
        register        int     m;      /* # records  in set 1 */
        register        int     n;      /* # records  in set 2 */
        register        int     i=0,j=0,terminate=0; /* loop counters  */
        int     copy1=0;

        top1 = bin_info[bin1].top;
        m = bin_info[bin1].size;
        top2 = bin_info[bin2].top;
        n = bin_info[bin2].size;

        while(! terminate)
        {
                if(keyComp(top1,top2) < 0)
                {
                        i++;
                        move(top1,topout,1);
                        topout+=kp;
                        top1+=kp;
                        if(i>=m)
                        {
                                terminate++;
                                copy1++;
                        }
                }
                else
                {
                        j++;
                        move(top2,topout,1);
                        top2 += kp;
                        topout += kp;
                        if(j >= n)
                                terminate++;
                }
        }
        if(copy1) move(top2,topout,(n-j));
        else move(top1,topout,(m-i));
        top1 -= (kp * i);
        topout = dc;
        move(topout,top1,m+n);          /* move back from buffer */
        bin_info[bin1].size += n;
}


move(from,to,numrecs)
register        char    *from,*to;
register        int     numrecs;        /* # records to move */
{
        register        int     i;
        register        int     reclen = kp;

        while(numrecs--)
        {
                i = reclen;
                while(i--)
                *(to++) = *(from++);
        }
}
```

```
/*******************************************************************/
/*                                                                 */
/* lsort2.c -- sortmerge assisted by cadm for filesize > cadmsize  */
/*                                                                 */
/* onlinesort is used for merge phase                              */
/*******************************************************************/

#include "header.h"
#define msec    0.0000625                /* 1/(16 MHz) in millisec       */

extern  int     bytesPerChip;
extern  int     useMask;                 /* true iff masks are used       */
extern  char    mask[512];               /* contains mask bytes if useMask*/
extern  char    d_a[MAXDATA];            /* list a for binary operations  */
extern  char    d_b[MAXDATA];            /* list b                        */
extern  char    dc_a[MAXDATA];           /* copy of list a for binary operations
*/
extern  char    dc_b[MAXDATA];           /* copy of list b                */
extern  int     k;                       /* # bytes in the key            */
extern  int     kp;                      /* # bytes in each record        */
extern  char    filea_name[10];
extern  char    fileb_name[10];
extern  long    dataSize_a;              /* # bytes in list a             */
extern  long    dataSize_b;              /* # bytes in list b             */
extern  long    dataSize;
extern  long    loadClocks;              /* time to load data             */
extern  long    sortClocks;              /* time to do sort               */
extern  long    searchClocks;            /* time to do searches           */
extern  long    broadcastClocks;         /* time to do broadcasts         */
extern  long    pushClocks;              /* time to do pushing            */
extern  long    totalClocks;             /* total time to sort data       */
        long    sloadClocks=0;           /* time to load data             */
        long    ssortClocks=0;           /* time to do sort               */
        long    stotalClocks=0;          /* total time to sort data       */
        long    ssearchClocks=0;         /* time to do searches           */
        long    sbroadcastClocks=0;      /* time to do broadcasts         */
        long    spushClocks=0;           /* time to do pushing            */
extern  char    *counter_buff;
extern  long    onlnsrt(),dataSize,pushTime();
char    *addrptr, *d;
int     softtime;
long    popclocks = 0;
float   onlnsrt_time = 0;


lsort2()
  {
     lsort3(d_a,dc_a,dataSize_a);
     foo();
  }



lsort3(dataary,copyary,dataSize_in)
char    *dataary, *copyary;
long    dataSize_in;
{
        long    tinputsize=0;   /* input size + tags                         */
        long    temp1,temp2,hi;
        long    i=0,j,l,m,a;
        long    fill_more=0;
        int     mergecount = 0;
        long    buff_size;

        struct bin_info_type
        {
                int     rec_in_bin;
                int     counter;        /* counter for merge              */
                int     rec_out;        /* # recs taken out               */
                char    *bin_top;       /* current top of the bin         */
        }bin_info[100];
        char    *cadm_buff;     /* top of cadm buffer for sortmerge      */
        int     recs_to_pop;    /* # records popped out per iteration    */
        int     numbins=1;      /* # bins for sort merge                 */
        char    *ptr,*from,*to,*calloc(),*ptr1;
        int     kpact;

        d = dataary;
        kpact = kp;
        kp++;
        bytesPerChip = ((1024 - kp) / kp ) * kp;
        dataSize = bytesPerChip * 16;   /* 16 K cadm space                */

        fprintf(stderr,"\nentering\n");
        fclose(stdout);

        for(i=0;i<100;i++)
                {
                        bin_info[i].rec_in_bin = 0;
                        bin_info[i].rec_out = 0;
                        bin_info[i].counter = 0;
                }

        i = 0;
        while(i < dataSize_in)  /* copy input into dc & append bin tag  */
        {
        bin_info[numbins-1].bin_top = &(copyary[tinputsize]);
        for(j=0;j < dataSize/kp && i < dataSize_in;j++)
        {
                for(l=1;l<kp;l++)
                        copyary[tinputsize++]=d[i++];
                copyary[tinputsize++]=numbins;  /* bin tag               */
                bin_info[numbins-1].rec_in_bin++;
        }
        numbins++;
        }
        numbins--;
        d = copyary;
        fprintf(stderr,"starting to sort each bin \n");
        for(i=0;i<numbins;i++)   /* sort each bin                         */
        {
        cadm(d,bin_info[i].rec_in_bin,kp,keyComp);
        d += ((bin_info[i].rec_in_bin)*kp);
        sloadClocks += loadClocks;
        ssortClocks += sortClocks;
        stotalClocks += totalClocks;
        ssearchClocks += searchClocks;
        sbroadcastClocks += broadcastClocks;
        spushClocks += pushClocks;
        loadClocks=0;
        sortClocks=0;
        totalClocks=0;
        searchClocks=0;
        broadcastClocks=0;
        pushClocks=0;
        }
        if(numbins < 4)
        stotalClocks = sloadClocks + (ssortClocks / numbins); /* when 4 independent 16K bl
ocks are used */
        else stotalClocks = sloadClocks + (ssortClocks / 4);
        fprintf(stderr,"finished sorting bins \n");
        buff_size = (bin_info[0].rec_in_bin);
```

```
cadm_buff = calloc((unsigned)(buff_size * kp),sizeof(char));
counter_buff = calloc((unsigned) (tinputsize / kp), sizeof(char));
if(cadm_buff == 0 || counter_buff == 0)
{
        fprintf(stderr,"not enough memory\n");
        exit(0);
}
ptr = cadm_buff;
i = (bin_info[0].rec_in_bin) / numbins;
for(m=0;m<numbins;m++)
{
        for(j=0;j<i && bin_info[m].rec_out < bin_info[m].rec_in_bin;j++)
        {
                for(l=0;l<kp;l++)
                *(ptr++) = *(bin_info[m].bin_top++);
                bin_info[m].rec_out++;
        }
        fill_more += (i-j);
}
fill_more += (bin_info[0].rec_in_bin) % numbins;
if(fill_more)
{
for(m=0;m<numbins - 1;m++)
{
        for(j=0;fill_more > 0
                && bin_info[m].rec_out < bin_info[m].rec_in_bin;j++)
        {
                for(l=0;l<kp;l++)
                *(ptr++) = *(bin_info[m].bin_top++);
                bin_info[m].rec_out++;
                fill_more--;
        }
}
}
cadm(cadm_buff,buff_size,kp,keyComp);
sloadClocks += loadClocks;
ssortClocks += sortClocks;
stotalClocks += totalClocks;
stotalClocks += sloadClocks;  /* for reading back each bin */
sloadClocks *= 2;       /* reading back each bin */
ssearchClocks += searchClocks;
sbroadcastClocks += broadcastClocks;
spushClocks += pushClocks;
loadClocks=0;
sortClocks=0;
totalClocks=0;
searchClocks=0;
broadcastClocks=0;
pushClocks=0;
m = 0;
recs_to_pop = dataSize / (kp * numbins);
d = cadm_buff;
ptr = cadm_buff;
ptr1 = counter_buff;
fprintf(stderr,"starting merge \n");
while( recs_to_pop )
{
        fprintf(stderr,"merging \n");
        ptr = cadm_buff;
        for(j=0;j<recs_to_pop;j++)
        {
         for(l=0;l<kpact && m < dataSize_in;l++)
                        dataary[m++] = *(ptr++);
        i = (int) *(ptr++);
        if(i < 100)bin_info[i - 1].counter++;
        *(ptr1++) = i;  /* save sequence of tags for timing */
```

```
                }
                popclocks += (recs_to_pop * pushTime(dataSize / bytesPerChip));
                i = recs_to_pop * kp;
                to = cadm_buff;
                from = cadm_buff+i;
                j = (buff_size * kp) - i;
                while(j--) *(to++) = *(from++);
                hi = to - cadm_buff;
                fill_more = 0;
                l = 0;
                recs_to_pop = 0;
                while(l++ < 2)  /* repeat twice to insert all records */
                {
                for(i=0;i<numbins;i++)
                {
                if(bin_info[i].rec_out == bin_info[i].rec_in_bin)
                        fill_more += bin_info[i].counter;
                else
                {
                        for(j=0;j < (bin_info[i].counter + fill_more)
                                && bin_info[i].rec_out < bin_info[i].rec_in_bin;j++)
                        {
                                onlnsrt_time += (msec * (onlinesort(cadm_buff,bin_info[i].
bin_top,hi)));

                                bin_info[i].bin_top += kp;
                                bin_info[i].rec_out++;
                                recs_to_pop++;
                                hi += kp;
                        }
                fill_more = (bin_info[i].counter + fill_more - j);
                }
                bin_info[i].counter = 0;
                }
            }
        if(fill_more) recs_to_pop = 0;
        }
        ptr = cadm_buff;
        fprintf(stderr,"finished merging\n");
        while(m < dataSize_in)
        {
                for(l=0;l<kpact;l++)
                        dataary[m++] = *(cadm_buff++);
                cadm_buff++;
        }

        softtime = mt(counter_buff,dataSize,dataSize_in); /*sw-time */

        stotalClocks += ( 6 * (cadm_buff - ptr));
        stotalClocks += onlnsrt_time / msec;
        stotalClocks += popclocks;
 }

foo()
   {
        FILE    *fpstat, *fpres, *fopen();

        fpres = fopen("result.out","w");
        fprintf(stderr,"writing output\n");
        d_a[dataSize_a] = '\0';
        fprintf(fpres,"%s",d_a);
        fclose(fpres);

        fpstat = fopen("result.stat","w");
        fprintf(fpstat,"\n\n\t\tCADM  SORT ENGINE  SIMULATOR  STATISTICS\n\n");
        fprintf(fpstat,"\t\tOperation :\t\tSortmerge\n");
        fprintf(fpstat,"\n\t\tFile :\t\t\t%s\n",filea_name);
        fprintf(fpstat,"\n\t\t# Records :\t\t%d\n",dataSize_a / (kp-1));
        fprintf(fpstat,"\n\t\tKey length : \t\t%d\n",k);
        fprintf(fpstat,"\n\t\tPointer length : \t%d\n",(kp-k-1));
        fprintf(fpstat, "\n\t\tTotal # Sort Clocks : \t%ld\n", ssortClocks);
        fprintf(fpstat, "\n\t\tTotal # Load Clocks : \t%ld\n", sloadClocks);
        fprintf(fpstat,"\n\t\tTotal # SON Clocks : \t%ld\n",onlnsrt_time / msec);
        fprintf(fpstat,"\n\t\tTotal # POP Clocks : \t%ld\n",popclocks);
        fprintf(fpstat, "\n\t\tTotal # Clock cycles :\t%ld\n", stotalClocks);
        fprintf(fpstat,"\n\t\tSort time : \t\t%f millisecs\n",ssortClocks * msec);
        fprintf(fpstat,"\n\t\tLoad time : \t\t%f millisecs\n",sloadClocks * msec);
        fprintf(fpstat,"\n\t\tSON  time : \t\t%f millisecs\n",onlnsrt_time);
        fprintf(fpstat,"\n\t\tPOP  time : \t\t%f millisecs\n",(popclocks * msec));
        fprintf(fpstat,"\n\t\tTotal CADM time : \t%f millisecs\n",stotalClocks  * msec);
        fprintf(fpstat,"\n\t\tSoftware Time : \t%d millisecs\n",softtime);
        fprintf(fpstat,"\n\t\tTotal time : \t\t%f milliseconds\n", ((float)softtime ) + st
otalClocks * msec);
        fprintf(fpstat,"\n\n\t\t\t*********************\n");
        fclose(fpstat);
 }
```

```
/*****************************************************************/
/*                                                               */
/* online sort simulation :                                      */
/* returns time required for an on line sort                     */
/*                                                               */
/*****************************************************************/

#include        "header.h"

long onlinesort(cadmbuf,inrecs,hi)
        char    *cadmbuf;       /* existing data in cadm        */
        char    *inrecs;        /* base of data to be inserted  */
        long    hi;             /* current end of data */
{
        long    insert_index;
        extern  long    dataSize,pushTime();
        long    findclocks,whereGoes;
        int     i;
        whereGoes = binarySearch2(inrecs,cadmbuf,hi);
        findclocks = fndTime(inrecs,hi);
        insert(cadmbuf,inrecs,whereGoes,hi);
        return(findclocks + (pushTime(dataSize / bytesPerChip)));
}

insert(cadmbuf,inrec,insert_index,hi)
        char    *cadmbuf,*inrec;
        long    insert_index;
        long    hi;             /* current end of data */
{
        int     i;
        long    move_bytes;
        char    *move_to,*insert_at;

        if(insert_index >= hi)

                for(i=0;i<kp;i++)
                        *(cadmbuf + hi + i) = *(inrec + i);
        else
        {
                move_bytes = hi - insert_index;
                insert_at = cadmbuf + insert_index;
                move_to = cadmbuf + hi + kp - 1;
                while(move_bytes >= 0)
                {
                        *(move_to) = *(move_to - kp);
                        move_to--;
                        move_bytes--;
                }
                for(i=0;i<kp;i++)
                *(insert_at++) = *(inrec++);
        }
        hi += kp;
}
```

```
/*********************************************************************/
/*                                                                   */
/* lsort2.c -- sortmerge assisted by cadm for filesize > cadmsize    */
/*                                                                   */
/* onlinesort is used for merge phase                                */
/*********************************************************************/

#include "header.h"
#define msec    0.0000833                 /* 1/(12 MHz) in millisec      */

extern  int     bytesPerChip;
extern  int     useMask;                  /* true iff masks are used      */
extern  char    mask[512];                /* contains mask bytes if useMask*/
extern  char    d_a[MAXDATA];             /* list a for binary operations  */
extern  char    d_b[MAXDATA];             /* list b                        */
extern  char    dc_a[MAXDATA];            /* copy of list a for binary operations
*/
extern  char    dc_b[MAXDATA];            /* copy of list b                */
extern  int     k;                        /* # bytes in the key            */
extern  int     kp;                       /* # bytes in each record        */
extern  char    filea_name[10];
extern  char    fileb_name[10];
extern  long    dataSize_a;               /* # bytes in list a             */
extern  long    dataSize_b;               /* # bytes in list b             */
extern  long    dataSize;
extern  long    loadClocks;               /* time to load data             */
extern  long    sortClocks;               /* time to do sort               */
extern  long    searchClocks;             /* time to do searches           */
extern  long    broadcastClocks;          /* time to do broadcasts         */
extern  long    pushClocks;               /* time to do pushing            */
extern  long    totalClocks;              /* total time to sort data       */
        long    sloadClocks=0;            /* time to load data             */
        long    ssortClocks=0;            /* time to do sort               */
        long    stotalClocks=0;           /* total time to sort data       */
        long    ssearchClocks=0;          /* time to do searches           */
        long    sbroadcastClocks=0;       /* time to do broadcasts         */
        long    spushClocks=0;            /* time to do pushing            */
extern  char    *counter_buff;
extern  long    onlnsrt(),dataSize,pushTime();
char    *addrptr, *d;
int     softtime;
long    popclocks = 0;
float   onlnsrt_time = 0;


lsort2()
  {
     lsort3(d_a,dc_a,dataSize_a);
     foo();
  }


lsort3(dataary,copyary,dataSize_in)
char    *dataary, *copyary;
long    dataSize_in;
{
        long    tinputsize=0;   /* input size + tags                     */
        long    temp1,temp2,hi;
        long    i=0,j,l,m,a;
        long    fill_more=0;
        int     mergecount = 0;
        long    buff_size;

        struct bin_info_type
        {
                int     rec_in_bin;
        int     counter;        /* counter for merge              */
        int     rec_out;        /* # recs taken out               */
        char    *bin_top;       /* current top of the bin         */
}bin_info[100];
char    *cadm_buff;     /* top of cadm buffer for sortmerge    */
int     recs_to_pop;    /* # records popped out per iteration  */
int     numbins=1;      /* # bins for sort merge               */
char    *ptr,*from,*to,*calloc(),*ptr1;
int     kpact;

d = dataary;
kpact = kp;
kp++;
bytesPerChip = ((1024 - kp) / kp ) * kp;
dataSize = bytesPerChip * 64;   /* 64 K cadm space             */

fprintf(stderr,"\nentering\n");
fclose(stdout);

for(i=0;i<100;i++)
        {
                bin_info[i].rec_in_bin = 0;
                bin_info[i].rec_out = 0;
                bin_info[i].counter = 0;
        }

i = 0;
while(i < dataSize_in)  /* copy input into dc & append bin tag  */
{
bin_info[numbins-1].bin_top = &(copyary[tinputsize]);
for(j=0;j < dataSize/kp && i < dataSize_in;j++)
{
        for(l=1;l<kp;l++)
                copyary[tinputsize++]=d[i++];
        copyary[tinputsize++]=numbins;  /* bin tag              */
        bin_info[numbins-1].rec_in_bin++;
}
numbins++;
}
numbins--;
d = copyary;
fprintf(stderr,"starting to sort each bin \n");
for(i=0;i<numbins;i++)   /* sort each bin                       */
{
cadm(d,bin_info[i].rec_in_bin,kp,keyComp);
d += ((bin_info[i].rec_in_bin)*kp);
sloadClocks += loadClocks;
ssortClocks += sortClocks;
stotalClocks += totalClocks;
ssearchClocks += searchClocks;
sbroadcastClocks += broadcastClocks;
spushClocks += pushClocks;
loadClocks=0;
sortClocks=0;
totalClocks=0;
searchClocks=0;
broadcastClocks=0;
pushClocks=0;
}
fprintf(stderr,"finished sorting bins \n");
buff_size = (bin_info[0].rec_in_bin);
cadm_buff = calloc((unsigned)(buff_size * kp),sizeof(char));
counter_buff = calloc((unsigned) (tinputsize / kp), sizeof(char));
if(cadm_buff == 0 || counter_buff == 0)
{
        fprintf(stderr,"not enough memory\n");
        exit(0);
}
ptr = cadm_buff;
i = (bin_info[0].rec_in_bin) / numbins;
for(m=0;m<numbins;m++)
{
        for(j=0;j<i && bin_info[m].rec_out < bin_info[m].rec_in_bin;j++)
        {
                for(l=0;l<kp;l++)
                *(ptr++) = *(bin_info[m].bin_top++);
                bin_info[m].rec_out++;
        }
        fill_more += (i-j);
}
fill_more += (bin_info[0].rec_in_bin) % numbins;
if(fill_more)
{
for(m=0;m<numbins - 1;m++)
{
        for(j=0;fill_more > 0
                && bin_info[m].rec_out < bin_info[m].rec_in_bin;j++)
        {
                for(l=0;l<kp;l++)
                *(ptr++) = *(bin_info[m].bin_top++);
                bin_info[m].rec_out++;
                fill_more--;
        }
}
}
cadm(cadm_buff,buff_size,kp,keyComp);
sloadClocks += loadClocks;
ssortClocks += sortClocks;
stotalClocks += totalClocks;
ssearchClocks += searchClocks;
sbroadcastClocks += broadcastClocks;
spushClocks += pushClocks;
stotalClocks += sloadClocks;  /* for reading back each bin */
sloadClocks *= 2;             /* reading back each bin */
loadClocks=0;
sortClocks=0;
totalClocks=0;
searchClocks=0;
broadcastClocks=0;
pushClocks=0;
m = 0;
recs_to_pop = dataSize / (kp * numbins);
d = cadm_buff;
ptr = cadm_buff;
ptr1 = counter_buff;
fprintf(stderr,"starting merge \n");
while( recs_to_pop )
{
        fprintf(stderr,"merging \n");
        ptr = cadm_buff;
        for(j=0;j<recs_to_pop;j++)
        {
         for(l=0;l<kpact && m < dataSize_in;l++)
                        dataary[m++] = *(ptr++);
        i = (int) *(ptr++);
        if(i < 100)bin_info[i - 1].counter++;
        *(ptr1++) = i;  /* save sequence of tags for timing */
        }
        popclocks += (recs_to_pop * pushTime(dataSize / bytesPerChip));
        i = recs_to_pop * kp;
        to = cadm_buff;
        from = cadm_buff+i;
```

```
                j = (buff_size * kp) - i;
                while(j--) *(to++) = *(from++);
                hi = to - cadm_buff;
                fill_more = 0;
                l = 0;
                recs_to_pop = 0;
                while(l++ < 2)  /* repeat twice to insert all records */
                {
                for(i=0;i<numbins;i++)
                {
                if(bin_info[i].rec_out == bin_info[i].rec_in_bin)
                        fill_more += bin_info[i].counter;
                else
                {
                        for(j=0;j < (bin_info[i].counter + fill_more)
                                && bin_info[i].rec_out < bin_info[i].rec_in_bin;j++)
                        {
                                onlnsrt_time += (msec * (onlinesort(cadm_buff,bin_info[i].
bin_top,hi)));

                                bin_info[i].bin_top += kp;
                                bin_info[i].rec_out++;
                                recs_to_pop++;
                                hi += kp;
                        }
                fill_more = (bin_info[i].counter + fill_more - j);
                }
                bin_info[i].counter = 0;
                }
            }
        if(fill_more) recs_to_pop = 0;
        }
        ptr = cadm_buff;
        fprintf(stderr,"finished merging\n");
        while(m < dataSize_in)
        {
                for(l=0;l<kpact;l++)
                        dataary[m++] = *(cadm_buff++);
                cadm_buff++;
        }

        softtime = mt(counter_buff,dataSize,dataSize_in); /*sw-time */

        stotalClocks += ( 6 * (cadm_buff - ptr));
        stotalClocks += onlnsrt_time / msec;
        stotalClocks += popclocks;
 }

foo()
   {
        FILE    *fpstat, *fpres, *fopen();

        /* fpres = fopen("result.out","w");
        fprintf(stderr,"writing output\n");
        d_a[dataSize_a] = '\0';
        fprintf(fpres,"%s",d_a);
        fclose(fpres); */

        fpstat = fopen("result.stat","a");
        fprintf(fpstat,"\n\n\t\tCADM  SORT ENGINE  SIMULATOR  STATISTICS\n\n");
        fprintf(fpstat,"\t\tOperation :\t\tSortmerge\n");
        fprintf(fpstat,"\n\t\tFile :\t\t\t%s\n",filea_name);
        fprintf(fpstat,"\n\t\t# Records :\t\t%d\n",dataSize_a / (kp-1));
        fprintf(fpstat,"\n\t\tKey length : \t\t%d\n",k);
        fprintf(fpstat,"\n\t\tPointer length : \t%d\n",(kp-k-1));
        fprintf(fpstat, "\n\t\tTotal # Sort Clocks : \t%ld\n", ssortClocks);
        fprintf(fpstat, "\n\t\tTotal # Load Clocks : \t%ld\n", sloadClocks);
        fprintf(fpstat,"\n\t\tTotal # SON Clocks : \t%ld\n",onlnsrt_time / msec);
        fprintf(fpstat,"\n\t\tTotal # POP Clocks : \t%ld\n",popclocks);
        fprintf(fpstat, "\n\t\tTotal # Clock cycles :\t%ld\n", stotalClocks);
        fprintf(fpstat,"\n\t\tSort time : \t\t%f millisecs\n",ssortClocks * msec);
        fprintf(fpstat,"\n\t\tLoad time : \t\t%f millisecs\n",sloadClocks * msec);
        fprintf(fpstat,"\n\t\tSON  time : \t\t%f millisecs\n",onlnsrt_time);
        fprintf(fpstat,"\n\t\tPOP  time : \t\t%f millisecs\n",(popclocks * msec));
        fprintf(fpstat,"\n\t\tTotal CADM time : \t%f millisecs\n",stotalClocks  * msec);
        fprintf(fpstat,"\n\t\tSoftware Time : \t%d millisecs\n",softtime);
        fprintf(fpstat,"\n\t\tTotal time : \t\t%f milliseconds\n", ((float)softtime ) + st
otalClocks * msec);
        fprintf(fpstat,"\n\n\t\t\t*********************\n");
        fclose(fpstat);
 }
```

```
/************************************************************************/
/*                                                                      */
/* this program gives the time required for the part of sortmerge that  */
/* is implemented on the host. i.e., the stuff that excludes operations */
/* using cadm.                                                          */
/*                                                                      */
/************************************************************************/

#include        "header.h"
#include        <sys/time.h>
#include        <sys/resource.h>
struct  rusage          startTime;
struct  rusage          endTime;

extern  int     bytesPerChip;
extern  int     useMask;                /* true iff masks are used
 */
extern  char    mask[512];              /* contains mask bytes if useMask*/
extern  char    d_a[MAXDATA];           /* list a for binary operations
  */
extern  char    d_b[MAXDATA];           /* list b
   */
extern  char    dc_a[MAXDATA];          /* copy of list a for binary operations
   */
extern  char    dc_b[MAXDATA];          /* copy of list b
    */
extern  int     k;                      /* # bytes in the key
     */
extern  int     kp;                     /* # bytes in each record
      */
extern  char    filea_name[10];
extern  char    fileb_name[10];
extern  long    dataSize_a;             /* # bytes in list a
       */
extern  long    dataSize_b;             /* # bytes in list b
        */
mt(counter_buff,dataSize,inputsize)
        char    *counter_buff;
        long    inputsize;              /* size of input */
        long    dataSize;               /* size of cadm memory */
{
        float   time;
        register int i;
        long    copyCount=0;
        long    sortCount=0;
        long    copyTime;
        long    sysCallTime;    /* time for a system call to check time */
        long    temp1,temp2;
        register int tinputsize;
        int     fill_more;
        int     buff_size;
        int     milliseconds;

        struct bin_info_type
        {
                int     rec_in_bin;
                int     counter;        /* counter for merge            */
                int     rec_out;        /* # recs taken out             */
                char    *bin_top;       /* current top of the bin       */
        }bin_info[100];
        char    *cadm_buff;     /* top of cadm buffer for sortmerge     */
        register int    recs_to_pop;    /* # records popped out per iteration   */
        register int    numbins=1;      /* # bins for sort merge        */
        register int    m=0,l,j;                /* for copying data */
        char    *ptr,*ptr1,*from,*to,*d,*calloc();
        FILE    *fopen(),*fp;
        tinputsize = 0;
        d = d_a;
        getrusage( RUSAGE_SELF, &startTime);
        endTime.ru_utime = startTime.ru_utime;
        while( endTime.ru_utime.tv_sec - startTime.ru_utime.tv_sec < 30 ) {
                for(i=0;i<100;i++)
                {
                        bin_info[i].rec_in_bin = 0;
                        bin_info[i].rec_out = 0;
                        bin_info[i].counter = 0;
                }
                numbins=1;      /* # bins for sort merge                */
                tinputsize = 0;
                i = 0;
                while(i < dataSize_a)   /* copy input into dc & append bin tag */
                {
                         bin_info[numbins-1].bin_top = &(dc_a[tinputsize]);
                         for(j=0;j < dataSize/kp && i < dataSize_a;j++)
                         {
                                for(l=1;l<kp;l++)
                                dc_a[tinputsize++]=d[i++];
                                dc_a[tinputsize++]=numbins;     /* bin tag */
                                bin_info[numbins-1].rec_in_bin++;
                         }
                         numbins++;
                 }
                numbins--;
                fill_more = 0;
                i = 0;
                l = 0;
                ptr = counter_buff;
                i = (int) *(ptr++);
                copyCount++;
                sortCount++;
                getrusage( RUSAGE_SELF, &endTime);
                }
        copyTime = 1000000 *
                ( endTime.ru_utime.tv_sec - startTime.ru_utime.tv_sec
) +
                ( endTime.ru_utime.tv_usec - startTime.ru_utime.tv_usec );
        copyTime += 1000000 *
                ( endTime.ru_stime.tv_sec - startTime.ru_stime.tv_sec
) +
                ( endTime.ru_stime.tv_usec - startTime.ru_stime.tv_usec );


        /* time the sort itself */
        sortCount = 0;
        tinputsize = 0;
        d = dc_a;

        getrusage( RUSAGE_SELF, &startTime);
        endTime.ru_utime = startTime.ru_utime;
        while( endTime.ru_utime.tv_sec - startTime.ru_utime.tv_sec < 30 ) {

        sortCount++;
                for(i=0;i<100;i++)
                {
                        bin_info[i].rec_in_bin = 0;
                        bin_info[i].rec_out = 0;
                        bin_info[i].counter = 0;
                }
        numbins=1;      /* # bins for sort merge                */
        fill_more = 0;
        i = 0;
        m = dataSize / kp;
        tinputsize = 0;
        while(i < dataSize_a)   /* copy input into dc & append bin tag
*/
         {
                 bin_info[numbins-1].bin_top = &(dc_a[tinputsize]);
                 for(j=0;j < m  && i < dataSize_a;j++)
                 {
                         for(l=1;l<kp;l++)
                         dc_a[tinputsize++]=d[i++];
                         dc_a[tinputsize++]=numbins;     /* bin tag */
                         bin_info[numbins-1].rec_in_bin++;
                 }
                 numbins++;
         }
        numbins--;
        for(i=0;i<numbins;i++);  /* sort each bin                    */
        m = (bin_info[0].rec_in_bin) / numbins;
        for(i=0;i<numbins;i++)
        {
                for(j=0;j<m && bin_info[i].rec_out < bin_info[i].rec_in_bin;j++)
                {
                        for(l=0;l<kp;l++)
                                bin_info[i].bin_top++;
                        bin_info[i].rec_out++;
                }
                fill_more += (m-j);
        }
        fill_more += (bin_info[0].rec_in_bin) % numbins;
        if(fill_more)
        {
        for(i=0;i<numbins - 1 && fill_more;i++)
        {
                for(j=0;fill_more > 0
                        && bin_info[i].rec_out < bin_info[i].rec_in_bin;j++)
                {
                        for(l=0;l<kp;l++)
                        bin_info[i].bin_top++;
                        bin_info[i].rec_out++;
                        fill_more--;
                }
        }
        }
        recs_to_pop = dataSize / (kp * numbins);
        ptr = counter_buff; /* from actual sort */
        while( recs_to_pop )
        {
                ptr1 = cadm_buff;
                for(j=0;j<recs_to_pop;j++)
                {
                i = (int) *(ptr++);
                bin_info[i - 1].counter++;
                }
                fill_more = 0;
                l = 0;
                recs_to_pop = 0;
                while(l++ < 2)  /* repeat twice to insert all records */
                {
                for(i=0;i<numbins;i++)
                {
                if(bin_info[i].rec_out == bin_info[i].rec_in_bin)
                        fill_more += bin_info[i].counter;
                else
                {
                        for(j=0;j < (bin_info[i].counter + fill_more)
                                && bin_info[i].rec_out < bin_info[i].rec_in_bin;j++)
                {
                        bin_info[i].bin_top += kp;
                        bin_info[i].rec_out++;
                        recs_to_pop++;
                }
        fill_more = (bin_info[i].counter + fill_more - j);
        }
        bin_info[i].counter = 0;
        }
}
if (fill_more) recs_to_pop = 0;
}
        getrusage( RUSAGE_SELF, &endTime);


        }  /* end while */


temp1 = 1000000 *
  ( endTime.ru_utime.tv_sec - startTime.ru_utime.tv_sec ) +
  ( endTime.ru_utime.tv_usec - startTime.ru_utime.tv_usec );
temp1 += 1000000 *      /* system */
  ( endTime.ru_stime.tv_sec - startTime.ru_stime.tv_sec ) +
  ( endTime.ru_stime.tv_usec - startTime.ru_stime.tv_usec );
temp1 = temp1 / sortCount;
temp2 = copyTime/copyCount;
milliseconds = (temp1 - temp2) / 1000;
return(milliseconds);

}       /* the end */
```

```
/**********************************************************************/
/*                                                                    */
/* lselect.c -- calculates the time necessary to find all occurences  */
/* of a given record.   i.e., returns time required to perform selection*/
/* on a list in cadm, when input size > cadm size (16k)                */
/*                                                                    */
/* April 9, 1987                                                      */
/*                                                                    */
/**********************************************************************/

#include        "header.h"
#include        <sys/time.h>
#include        <sys/resource.h>


#define MICROSEC        0.0625          /* 1/(16MHz) in microsec        */
#define msec    0.0000625               /* 1/(16 MHz) in millisec      */

extern  long    fndTime();
extern  long    fndInitTime();


extern  int     bytesPerChip;
extern  int     useMask;                /* true iff masks are used      */
extern  char    mask[512];              /* contains mask bytes if useMask*/
extern  char    *d;
extern  char    d_a[MAXDATA];           /* list a for binary operations  */
extern  char    d_b[MAXDATA];           /* list b                        */
extern  char    dc_a[MAXDATA];          /* copy of list a for binary operations */
extern  char    dc_b[MAXDATA];          /* copy of list b                */
extern  int     k;                      /* # bytes in the key            */
extern  int     kp;                     /* # bytes in each record        */
extern  char    filea_name[10];
extern  char    fileb_name[10];
extern  long    dataSize_a;             /* # bytes in list a             */
extern  long    dataSize_b;             /* # bytes in list b             */
extern  long    dataSize;
char    *addrptr;
extern  char    pred;           /* selection criterion :
                                        =  ==  a
                                        >  ==  b
                                        >= ==  c
                                        <  ==  d
                                        <= ==  e
                                  range     ==  f  */

extern  long    stotalClocks;           /* total time to sort data       */
extern  int     softtime;

lselect()
{
        float   time;
        FILE    *fopen();
        FILE    *fpout, *fpstat;
        long    i;
        long    copyCount;
        long    copyTime;
        long    sortCount;
        long    temp1,temp2;
        long    dataSize;
        long    milliseconds;
        char    *ptr;
        char    *list;
        int     nofind;
        char    *upper;
        char    *lower;
        register        int     numrecs;        /* # records to be found (selected ) */
        register        int     count;
        register        int     z;
        char    inLine[256];
        char    inLine1[256];
        int     status;         /* status pin of the cadm */
        int     done;           /* done pin of the cadm */
        int     stop;
long    float   totaltime=0.0;
struct  rusage          startTime;
struct  rusage          endTime;


        lsort3(d_a,dc_a,dataSize_a);            /* sort input data and put result in d_a[]
 */

        totaltime += softtime;
        totaltime += (msec * stotalClocks);
        kp--;                   /* take off the tag byte from record*/
        d = d_a;
        dataSize = dataSize_a;

        /* reopen the to-find file */
        if ( fileb_name[0] == '\0' ) {
                freopen( "/dev/tty", "r", stdin);
                }
        else {
                if (freopen( fileb_name, "r", stdin) == NULL) {
                        fprintf( stdout, "cannot open %s\n", fileb_name);
                        exit( 1);
                        }
                }

        fpout = fopen("result.out","w");
        gets( inLine);
        gets( inLine1);
        if ( strlen(inLine) != k )
        printf("Your key \"%s\" is not %d bytes long.\n", inLine,k);




        sortCount = 0;
        getrusage( RUSAGE_SELF, &startTime);
        endTime.ru_utime = startTime.ru_utime;
        while( endTime.ru_utime.tv_sec - startTime.ru_utime.tv_sec < 30 )  {

        switch(pred)
        {
            case 'a' :          /*  ==   */
            ptr = d_b;
               upper =  d + binarySearch2(inLine,d,dataSize);
               nofind = keyComp(inLine, upper);
               while (! nofind )
               {
                  for(z=0;z<kp;z++) *ptr++ = *upper++;
                  nofind = keyComp(inLine, upper);
                }
              *ptr = '\0';
            break;

            case 'b' :          /*  >  */
                upper =  d + binarySearch(inLine,d,dataSize);
                numrecs = (d + dataSize) - upper;
                for(count = 0; count < numrecs; count++)
                  d_b[count] = *upper++;
            d_b[count] = '\0';

        break;

        case 'c' :           /*  >=  */
            upper =  d + binarySearch2(inLine,d,dataSize);
            numrecs = (d + dataSize) - upper;
            for(count = 0; count < numrecs; count++)
              d_b[count] = *upper++;
            d_b[count] = '\0';

        break;

        case 'd' :           /* < */
            upper =  d + binarySearch2(inLine,d,dataSize);
            numrecs = upper - d;
            upper = d;
            for(count = 0; count < numrecs; count++)
              d_b[count] = *upper++;
            d_b[count] = '\0';

        break;

        case 'e' :           /* <= */
            upper =  d + binarySearch(inLine,d,dataSize);
            numrecs = upper - d;
            upper = d;
            for(count = 0; count < numrecs; count++)
              d_b[count] = *upper++;
            d_b[count] = '\0';

        break;

        case 'f' :           /* <> i.e., > 1st key, < 2nd key*/
            lower =  d + binarySearch(inLine,d,dataSize);
            upper =  d + binarySearch2(inLine1,d,dataSize);
            numrecs = upper - lower;
            for(count = 0; count < numrecs; count++)
              d_b[count] = *lower++;
            d_b[count] = '\0';

    }


    sortCount++;
    getrusage( RUSAGE_SELF, &endTime);
}  /* end while */
    temp1 = 1000000 *
    ( endTime.ru_utime.tv_sec - startTime.ru_utime.tv_sec ) +
    ( endTime.ru_utime.tv_usec - startTime.ru_utime.tv_usec );
    temp1 += 1000000 *      /* system */
    ( endTime.ru_stime.tv_sec - startTime.ru_stime.tv_sec ) +
    ( endTime.ru_stime.tv_usec - startTime.ru_stime.tv_usec );
    temp1 = temp1 / sortCount;
    milliseconds = temp1 / 1000;


    fpout = fopen("result.out","w");         /* output file */
    fpstat = fopen("result.stat","w");       /* statistics */

    fprintf(fpout,"%s",d_b);

    fprintf(fpstat,"\n\n\t\tCADM  SORT ENGINE  SIMULATOR  STATISTICS\n\n");
    fprintf(fpstat,"\t\tOperation :\t\tSelection\n\n");
    fprintf(fpstat,"\n\t\tInput File :\t\t%s\n",filea_name);
    fprintf(fpstat,"\n\t\tRecords File :\t\t%s\n",fileb_name);
```

```
        fprintf(fpstat,"\n\t\t# Records :\t\t%d\n",dataSize_a / kp);
        fprintf(fpstat,"\n\t\tKey length : \t\t%d\n",k);
        fprintf(fpstat,"\n\t\tPointer length : \t%d\n",kp-k);
        fprintf(fpstat,"\n\t\tCADM Sortmerge time : \t%lf millisecs\n",totaltime);
        fprintf(fpstat,"\n\t\tTotal time : \t\t%lf millisecs\n",(float) (totaltime + milli
seconds));
        fprintf(fpstat,"\n\n\t\t\t**********************\n");
        fclose(fpout);
        fclose(fpstat);

}
```

```
/*************************************************************************/
/*                                                                       */
/*  ldiff.c      : finds  the difference of two lists and returns        */
/*  the time required.when the input size is greater than the CADM size.*/
/*                                                                       */
/*                                                                       */
/* April 2, 1987                                                         */
/*                                                                       */
/*************************************************************************/


#include        "header.h"
#include        <sys/time.h>
#include        <sys/resource.h>

#define MICROSEC        0.0625  /* 1/(16MHz) in m-sec                    */
#define msec    0.0000625               /* 1/(16 MHz) in millisec       */


extern  long    fndTime();
extern  long    fndInitTime();


extern  int     bytesPerChip;
extern  int     useMask;                /* true iff masks are used       */
extern  char    mask[512];              /* contains mask bytes if useMask*/
extern  char    *d;
extern  char    d_a[MAXDATA];           /* list a for binary operations  */
extern  char    d_b[MAXDATA];           /* list b                        */
extern  char    dc_a[MAXDATA];          /* copy of list a for binary operations
*/
extern  char    dc_b[MAXDATA];          /* copy of list b                */
extern  int     k;                      /* # bytes in the key            */
extern  int     kp;                     /* # bytes in each record        */
extern  char    filea_name[10];
extern  char    fileb_name[10];
extern  long    dataSize_a;             /* # bytes in list a             */
extern  long    dataSize_b;             /* # bytes in list b             */
extern  long    dataSize;


extern  long    loadClocks;
extern  long    broadcastClocks;
extern  long    searchClocks;
extern  long    pushClocks;
extern  long    sortClocks;
extern  long    totalClocks;
extern  int     softtime;
char    *addrptr;

ldifference()
{
        char    *skey;
        long    i;              /* for copying data */
        int     status;         /* status pin of the cadm */
        int     done;           /* done pin of the cadm */
        int     numrecs;        /* # records in the smaller list in cadm */
        int     numbins;        /* # bins in the smaller list in cadm */
        FILE    *fp,*fpout,*fpstat;
        long    load_a;
        long    sort_a;
        long    load_b;
        long    sort_b;
        char    *ptr;
        char    *result;        /* result of intersection */
        long    size;
        char    *malloc();
        char    *list;
        int     flag = 0;
        long    stotalClocks=0;
        int     ssofttime=0;
        long    diffClocks=0;
        int     count=0;
        int     last_bin_size;
        extern  long    pop();


/*  sort each input list  */

        fclose(stdout);
        if(dataSize_a < 16000)
           cadm(d_a,(int)dataSize_a / kp,kp,keyComp);
        else
         {
           lsort3(d_a,dc_a,dataSize_a); /* sort list a */
           kp--;
         }

        stotalClocks += totalClocks;
        ssofttime += softtime;
        softtime = 0;
        loadClocks = 0;
        broadcastClocks = 0;
        searchClocks = 0;
        pushClocks = 0;
        sortClocks = 0;
        if(dataSize_b < 16000)
           cadm(d_b,(int)dataSize_b / kp,kp,keyComp);
        else
         {
           lsort3(d_b,dc_b,dataSize_b); /* sort list a */
           kp--;
         }
        stotalClocks += totalClocks;
        ssofttime += softtime;

        bytesPerChip = ((1024 - kp) / kp ) * kp;
        dataSize = bytesPerChip * 16;   /* 16 K cadm space */

             d = d_a;
             ptr  = d_b;
             numbins = dataSize_a / dataSize;
             numrecs = dataSize_b / kp;
             if(dataSize_a % dataSize)
              {
                flag++;
                numbins++;
                last_bin_size = dataSize_a % dataSize;
              }
             if(dataSize > dataSize_a)dataSize = dataSize_a;


 /* now, d points to beginning of list a in RAM, which has numbins bins*/
 /* list points to list  b in RAM  with numrecs records.*/

        skey = malloc(k);

        result = dc_a;
        done = 0;
        while(numbins-- && numrecs)
        {
                diffClocks += (6 * dataSize); /* for loading bin into CADM */
                size = dataSize;
                done = 0;
                while(numrecs-- && !done)
                {
                  strncpy(skey,ptr,k);
                  diffClocks += fndTime(skey, size);
                  diffClocks += 8 * k;   /* for fetching key from list in RAM */
                  addrptr = d + binarySearch2(skey,d,size);
                  done = (addrptr >= (d + size));
                  status = keyComp(skey, addrptr);
                  if(! status)  /* if record does exist in large list */
                     {
                        diffClocks += pop(addrptr, d, size);  /* pop record out */

                        size -= kp;
                     }
                   ptr += kp;
                }
                numrecs++;
                for(i=0;i<size;i++) *result++ = d[i];
                diffClocks += 6 * size;
                d += dataSize;

                if(numbins == 1)
                  if(flag)
                      dataSize = last_bin_size;
        }


fp = fopen("result.out","w");
*result = '\0';
fprintf(fp,"%s",dc_a);
fpstat = fopen("result.stat","w");      /* statistics */

fprintf(fpstat,"\n\n\t\tCADM  SORT ENGINE  SIMULATOR  STATISTICS\n\n");
fprintf(fpstat,"\t\tOperation :\t\tDifference\n");
fprintf(fpstat,"\n\t\tFile_A :\t\t%s\n",filea_name);
fprintf(fpstat,"\n\t\tFile_B :\t\t%s\n",fileb_name);
fprintf(fpstat,"\n\t\t# Records_A :\t\t%d\n",dataSize_a / kp);
fprintf(fpstat,"\n\t\t# Records_B :\t\t%d\n",dataSize_b / kp);
fprintf(fpstat,"\n\t\tKey length : \t\t%d\n",k);
fprintf(fpstat,"\n\t\tPointer length : \t%d\n",kp-k);
fprintf(fpstat,"\n\t\tSort Time : \t\t%f millisecs\n",(stotalClocks * msec) + ssofttime);
fprintf(fpstat,"\n\t\tDifference Time : \t%f millisecs\n",(diffClocks * msec));
fprintf(fpstat,"\n\t\tTotal time : \t\t%f millisecs\n",(ssofttime + (diffClocks + stotalCl
ocks )* msec));
fprintf(fpstat,"\n\n\t\t\t********************\n");
fclose(fp);
fclose(fpstat);

}  /* end of main */
```

```
/*********************************************************************/
/*                                                                   */
/*  lintersect.c      : finds  the intersection of two lists and returns */
/*  the time required.when the input size is greater than the CADM size.*/
/*                                                                   */
/*                                                                   */
/* April 2, 1987                                                     */
/*                                                                   */
/*********************************************************************/


#include        "header.h"
#include        <sys/time.h>
#include        <sys/resource.h>

#define MICROSEC         0.0625  /* 1/(16MHz) in m-sec                    */
#define msec    0.0000625               /* 1/(16 MHz) in millisec      */


extern  long    fndTime();
extern  long    fndInitTime();


extern  int     bytesPerChip;
extern  int     useMask;                /* true iff masks are used       */
extern  char    mask[512];              /* contains mask bytes if useMask*/
extern  char    *d;
extern  char    d_a[MAXDATA];           /* list a for binary operations  */
extern  char    d_b[MAXDATA];           /* list b                        */
extern  char    dc_a[MAXDATA];          /* copy of list a for binary operations
*/
extern  char    dc_b[MAXDATA];          /* copy of list b                */
extern  int     k;                      /* # bytes in the key            */
extern  int     kp;                     /* # bytes in each record        */
extern  char    filea_name[10];
extern  char    fileb_name[10];
extern  long    dataSize_a;             /* # bytes in list a             */
extern  long    dataSize_b;             /* # bytes in list b             */
extern  long    dataSize;


extern  long    loadClocks;
extern  long    broadcastClocks;
extern  long    searchClocks;
extern  long    pushClocks;
extern  long    sortClocks;
extern  long    totalClocks;
extern  int     softtime;
char    *addrptr;

lintersect()
{
        char    *skey;
        long    i;              /* for copying data */
        int     status;         /* status pin of the cadm */
        int     done;           /* done pin of the cadm */
        int     numrecs;        /* # records in the smaller list in cadm */
        int     numbins;        /* # bins in the smaller list in cadm */
        FILE    *fp,*fpout,*fpstat;
        long    load_a;
        long    sort_a;
        long    load_b;
        long    sort_b;
        char    *ptr;
        char    *result;        /* result of intersection */
        long    total;          /* total clockcycles required */
        char    *malloc();
        char    *list;
        int     flag = 0;
        long    stotalClocks=0;
        int     ssofttime=0;
        long    unionClocks=0;
        float   usofttime = 0.0;
        int     count=0;
        int     last_bin_size;
        extern  float   i_time();


/*  sort each input list  */

        fclose(stdout);
        if(dataSize_a < 16000)
           cadm(d_a,(int)dataSize_a / kp,kp,keyComp);
        else
         {
           lsort3(d_a,dc_a,dataSize_a); /* sort list a */
           kp--;
         }

        stotalClocks += totalClocks;
        ssofttime += softtime;
        softtime = 0;
        loadClocks = 0;
        broadcastClocks = 0;
        searchClocks = 0;
        pushClocks = 0;
        sortClocks = 0;
        if(dataSize_b < 16000)
           cadm(d_b,(int)dataSize_b / kp,kp,keyComp);
        else
         {
           lsort3(d_b,dc_b,dataSize_b); /* sort list a */
           kp--;
         }
        stotalClocks += totalClocks;
        ssofttime += softtime;

        bytesPerChip = ((1024 - kp) / kp ) * kp;
        dataSize = bytesPerChip * 16;   /* 16 K cadm space */

        if(dataSize_b / kp < dataSize_a / kp)
          {
             d = d_b;
             list  = d_a;
             numbins = dataSize_b / dataSize;
             numrecs = dataSize_a / kp;
             if(dataSize_b % dataSize)
              {
                flag++;
                numbins++;
                last_bin_size = dataSize_b % dataSize;
              }
             if(dataSize > dataSize_b)dataSize = dataSize_b;
          }
        else
          {
             d = d_a;
             list  = d_a;
             list = d_b;
             numbins = dataSize_a / dataSize;
             numrecs = dataSize_b / kp;
```

```
            if(dataSize_a % dataSize)
             {
               flag++;
               numbins++;
               last_bin_size = dataSize_a % dataSize;
             }
            if(dataSize > dataSize_a)dataSize = dataSize_a;
          }



/* now, d points to beginning of the smaller list in RAM, which has*/
/* numbins bins.list points to larger list in RAM  with numrecs records.*/

        total = 0;
        skey = malloc(k);

        ptr = list;
        result = list;
        done = 0;

        while(numbins-- && numrecs)
        {
                unionClocks += (12 * dataSize); /* for loading bin into CADM and for takin
g it out later */
                done = 0;
                while(numrecs-- && !done)
                {
                  strncpy(skey,ptr,k);
                  unionClocks += fndTime(skey, dataSize);
                  unionClocks += 8 * k;   /* for fetching key from list in RAM */
                  addrptr = d + binarySearch2(skey,d,dataSize);
                  status = keyComp(skey, addrptr);
                  done = (addrptr >= (d + dataSize));
                  if(!status)
                    {
                      for(i=0;i<kp;i++) *result++ = *ptr++;
                      unionClocks += 8 * kp;    /* for reading record back to answer buffe
r */
                    }
                  else ptr += kp;
                }
                numrecs++;
                if(numbins > 1)
                  d += dataSize;
                else
                {
                  if(flag)
                    {
                      d += last_bin_size;
                      dataSize = last_bin_size;
                    }
                }
        }


 fp = fopen("result.out","w");
 ptr = list;
 for(i=0;i<result - list;i++) fprintf(fp,"%c",*ptr++);
 fpstat = fopen("result.stat","w");      /* statistics */

 fprintf(fpstat,"\n\n\t\tCADM  SORT ENGINE  SIMULATOR  STATISTICS\n\n");
 fprintf(fpstat,"\t\tOperation :\t\tIntersection\n");
 fprintf(fpstat,"\n\t\tFile_A :\t\t%s\n",filea_name);
 fprintf(fpstat,"\n\t\tFile_B :\t\t%s\n",fileb_name);
 fprintf(fpstat,"\n\t\t# Records_A :\t\t%d\n",dataSize_a / kp);
fprintf(fpstat,"\n\t\t# Records_B :\t\t%d\n",dataSize_b / kp);
fprintf(fpstat,"\n\t\tKey length : \t\t%d\n",k);
fprintf(fpstat,"\n\t\tPointer length : \t%d\n",kp-k);
fprintf(fpstat,"\n\t\tSort Time : \t\t%f millisecs\n",(stotalClocks * msec) + ssofttime);
fprintf(fpstat,"\n\t\tIntersection Time : \t%f millisecs\n",(unionClocks * msec));
fprintf(fpstat,"\n\t\tTotal time : \t\t%f millisecs\n",(ssofttime + (unionClocks + stotalC
locks )* msec));
fprintf(fpstat,"\n\n\t\t\t*********************\n");
fclose(fp);
fclose(fpstat);

}  /* end of main */
```

```
/***********************************************************************/
/*                                                                     */
/*  This file contains the program for timing the sort engine operations */
/*  performed entirely on SUN 3/50 for comparison purposes.            */
/*                                                                     */
/*  Performs :                                                         */
/*                                                                     */
/*              Sorting using Brian's quicksort,                       */
/*              Selection ( >, >=, ==, <, <=),                         */
/*              Intersection,                                          */
/*              Union  and                                             */
/*              Set difference.                                        */
/*                                                                     */
/*  Author :  Vivekanand Bhat.                                         */
/*                                                                     */
/*  January  13, 1987                                                  */
/*                                                                     */
/***********************************************************************/


#include "header.h"

char    d_a[MAXDATA];           /* list a for binary operations       */
char    d_b[MAXDATA];           /* list b                             */
char    dc_a[MAXDATA];          /* copy of list a for binary operations */
char    dc_b[MAXDATA];          /* copy of list b                     */
int     useMask;                /* true iff masks are used           */
char    mask[512];              /* contains mask bytes if useMask    */
int     k;                      /* # bytes in the key                 */
int     kp;                     /* # bytes in each record             */
FILE    *fp;
char    pred;                   /* predicate for select              */
char    filea_name[10];
char    fileb_name[10];
long    dataSize_a;             /* # bytes in list a                  */
long    dataSize_b;             /* # bytes in list b                  */
long    dataSize;
long    bytesPerChip;

main(argc, argv)
{

   struct optype
    {
       char     op_name[10];    /* name of the operation              */
       int      (*function)();  /* pointer to the function            */
    }op[5];
   char *strcpy();
   char command[10];            /* command from user                 */
   extern       int     quicksort();/* brian's quicksort             */
   extern       int     select();       /* performs selection        */
   extern       int     intersection();/* preforms intersection      */
   extern       int     un();   /* preforms union                    */
   extern       int     difference();/* preforms set difference      */

   long i;

   strcpy(op[0].op_name, "sort");       /* initialize ops */
   strcpy(op[1].op_name, "select");
   strcpy(op[2].op_name, "intersect");
   strcpy(op[3].op_name, "union");
   strcpy(op[4].op_name, "diff");

   op[0].function = quicksort;
   op[1].function = select;
```

```
    op[2].function = intersection;
    op[3].function = un;
    op[4].function = difference;

    scancommand(argc, argv, command);    /* scan input command line */

    fp = fopen (filea_name, "r");
    if(fp == '\0')
      {
        fprintf(stderr,"\nCannot open file %s\n",filea_name);
        exit(-1);
      }
    dataSize_a = getinput( d_a, fp);
    fclose (fp);

    if(fileb_name[0] != '\0')
    {
      fp = fopen (fileb_name, "r");
      if(fp == '\0')
        {
          fprintf(stderr,"\nCannot open file %s\n",fileb_name);
          exit(-1);
        }
      dataSize_b = getinput( d_b, fp);
      fclose (fp);
    }

    for(i=0;i<5 && ((strcmp(op[i].op_name,command))!=0);i++); /* what command? */
    if(i == 5)             /* oops, wrong one */
      {
        fprintf(stderr, "\nCommand %s not found\n",command);
        exit(-1);
      }
    (*op[i].function)(); /* execute appropriate function                  */

    fprintf(stderr,"\n Result written to file result.out \n");
    fprintf(stderr,"\n Statistics written to file result.stat \n");


}



/* getinput -- read input file and check format                          */




    long
    getinput( d, fp)
    char    *d;             /* where to put the data                   */
    FILE    *fp;            /* pointer to input file                   */
    {
    int     c;
    long    dataSize;

     dataSize = 0;
     while ( (c=fgetc(fp)) != EOF )
     {
       d[ dataSize++] = c;
       if ( dataSize >= MAXDATA ) {
       fprintf( stdout, "too much data--change MAXDATA\n");
       exit( -1);
       }
       if ( c == 0 ) {
       fprintf( stdout, "%s%s%s",
      "The c language is picky about binary zero's.\n",
      "Please remove them from your input file.\n",
      "(The CADM has no problems with them.)\n");
      exit( -1);
      }
    }
    if ( dataSize % kp != 0 && (dataSize % (k+1))!=0) {
        fprintf( stdout, "input is not integral number of records\n");
        exit( -1);
        }

        return( dataSize);
}
```

```
/***********************************************************************/
/*                                                                     */
/* sw_diff.c -- calculates the time necessary to find difference       */
/* of a two lists on a SUN 3/50                                        */
/*                                                                     */
/*                                                                     */
/* January  9, 1987                                                    */
/*                                                                     */
/***********************************************************************/

#include        "header.h"
#include        <sys/time.h>
#include        <sys/resource.h>


#define MICROSEC        0.0625  /* 1/(16MHz) in m-sec                  */


extern  char    d_a[MAXDATA];           /* list a for binary operations         */
extern  char    d_b[MAXDATA];           /* list b                               */
extern  char    dc_a[MAXDATA];          /* copy of list a for binary operations */
extern  char    dc_b[MAXDATA];          /* copy of list b                       */
extern  int     useMask;                /* true iff masks are used             */
extern  char    mask[512];              /* contains mask bytes if useMask      */
extern  int     k;                      /* # bytes in the key                   */
extern  int     kp;                     /* # bytes in each record               */
extern  char    filea_name[10];
extern  char    fileb_name[10];
extern  long    dataSize_a;             /* # bytes in list a                    */
extern  long    dataSize_b;             /* # bytes in list b                    */
extern  long    binarySearch();
extern  long    binarySearch2();
struct  rusage          startTime;
struct  rusage          endTime;
char    result[MAXDATA]; /* result of intersection */


difference()
{

        float   time;
        FILE    *fopen();
        FILE    *fpout, *fpstat;
        register long    i;
        long    copyCount;
        long    copyTime;
        long    sortTime;
        long    diffCount;
        long    diffTime;
        long    sortCount;
        register int    bytesout;
        long    temp1,temp2,temp3;
        long    dataSize;
        long    milliseconds;
        register char    *d;
        register char    *ptr;
        register char    *addrptr;
        register char    *ptr1;
        char    *list;
        register char   *skey;
        char    *strncpy();
        register int    noFind;
        register int     done;
        char    *upper;
        char    *lower;
        char    *malloc();
        register        int     numrecs;        /* # records to be found (selected ) */

        sortCount = 0;
        copyCount = 0;
        diffCount = 0;


        ptr = dc_a;
        skey = malloc(k);
        upper = result;

        /* time to copy */
        getrusage( RUSAGE_SELF, &startTime);
        endTime.ru_utime = startTime.ru_utime;
        while( endTime.ru_utime.tv_sec - startTime.ru_utime.tv_sec < 30 ) {
                for( i=0; i < dataSize_a; i++) {
                dc_a[i] = d_a[i];
        }
                for( i=0; i < dataSize_b; i++) {
                dc_b[i] = d_b[i];
        }
        strncpy(skey,ptr,k);
        copyCount++;
        getrusage( RUSAGE_SELF, &endTime);
        }
        copyTime = 1000000 *
        ( endTime.ru_utime.tv_sec - startTime.ru_utime.tv_sec ) +
        ( endTime.ru_utime.tv_usec - startTime.ru_utime.tv_usec );
        copyTime += 1000000 *   /* system */
        ( endTime.ru_stime.tv_sec - startTime.ru_stime.tv_sec ) +
        ( endTime.ru_stime.tv_usec - startTime.ru_stime.tv_usec );


        getrusage( RUSAGE_SELF, &startTime);
        endTime.ru_utime = startTime.ru_utime;
        while( endTime.ru_utime.tv_sec - startTime.ru_utime.tv_sec < 30 ) {
                for( i=0; i < dataSize_a; i++) {
                dc_a[i] = d_a[i];
                }
                for( i=0; i < dataSize_b; i++) {
                dc_b[i] = d_b[i];
                }
                trosq(d_a,(int)dataSize_a/kp,kp,keyComp);
                trosq(d_b,(int)dataSize_b/kp,kp,keyComp);
        sortCount++;
        getrusage( RUSAGE_SELF, &endTime);
        }
        sortTime = 1000000 *
        ( endTime.ru_utime.tv_sec - startTime.ru_utime.tv_sec ) +
        ( endTime.ru_utime.tv_usec - startTime.ru_utime.tv_usec );
        sortTime += 1000000 *   /* system */
        ( endTime.ru_stime.tv_sec - startTime.ru_stime.tv_sec ) +
        ( endTime.ru_stime.tv_usec - startTime.ru_stime.tv_usec );


        getrusage( RUSAGE_SELF, &startTime);
        endTime.ru_utime = startTime.ru_utime;
        while( endTime.ru_utime.tv_sec - startTime.ru_utime.tv_sec < 30 ) {

            d = d_a;
            list = d_b;
            dataSize = dataSize_a;
            numrecs = dataSize_b / kp;

    /* now, d points to list a, dataSize is its size,  */
```

```
/*list points to list b  with numrecs records.      */


         ptr = list;
         ptr1 = upper;
         done = 0;


         while(numrecs && !done)
          {
              numrecs--;
              strncpy(skey,ptr,k);
              addrptr = d + binarySearch2(skey,d,dataSize);
              noFind = keyComp(skey, addrptr);
              done = (addrptr >= (d + dataSize));
              if(!noFind)
                {
                 bytesout = addrptr - d;
                 for(i=0;i<bytesout;i++) *ptr1++ = *d++;
                 d += kp;
                 dataSize -= bytesout;
                 dataSize -= kp;
                }
               ptr += kp;
          }


        diffCount++;
        getrusage( RUSAGE_SELF, &endTime);
  }  /* end while */
        temp1 = 1000000 *
        ( endTime.ru_utime.tv_sec - startTime.ru_utime.tv_sec ) +
        ( endTime.ru_utime.tv_usec - startTime.ru_utime.tv_usec );
        temp1 += 1000000 *       /* system */
        ( endTime.ru_stime.tv_sec - startTime.ru_stime.tv_sec ) +
        ( endTime.ru_stime.tv_usec - startTime.ru_stime.tv_usec );
        temp1 = temp1 / diffCount;
        temp2 = copyTime / copyCount;   /* time for copy and sys call*/
        temp3 = sortTime / sortCount;   /* time for sort*/
        milliseconds = (temp1  + temp3 - temp2) / 1000;


        fpout = fopen("result.out","w");        /* output file */
        fpstat = fopen("result.stat","w");      /* statistics */

        *ptr1 = '\0';
        fprintf(fpout,"%s",result);

        fprintf(fpstat,"\n\n\t\tSORT ENGINE  SIMULATOR  STATISTICS\n\n");
        fprintf(fpstat,"\t\t\t\tSUN 3/50\n\n");
        fprintf(fpstat,"\t\tOperation :\t\tDifference\n\n");
        fprintf(fpstat,"\n\t\tInput File :\t\t%s\n",filea_name);
        fprintf(fpstat,"\n\t\tRecords File :\t\t%s\n",fileb_name);
        fprintf(fpstat,"\n\t\t# Records_a :\t\t%d\n",dataSize_a / kp);
        fprintf(fpstat,"\n\t\t# Records_b :\t\t%d\n",dataSize_b / kp);
        fprintf(fpstat,"\n\t\tKey length : \t\t%d\n",k);
        fprintf(fpstat,"\n\t\tPointer length : \t%d\n",kp-k);
        fprintf(fpstat,"\n\t\tSort time : \t\t%ld millisecs\n",(temp3 - temp2) / 1000);
        fprintf(fpstat,"\n\t\tDifference time : \t%ld millisecs\n",temp1 / 1000);
        fprintf(fpstat,"\n\t\tTotal time : \t\t%ld millisecs\n",milliseconds);
        fprintf(fpstat,"\n\n\t\t\t********************\n");
        fclose(fpout);
        fclose(fpstat);

}  /* end of main */
```

```
/**********************************************************************/
/*                                                                    */
/* sw_intersection.c -- calculates the time necessary to find intersection      */
/* of a given list with another list on a SUN 3/50                    */
/*                                                                    */
/*                                                                    */
/* January  9, 1987                                                   */
/*                                                                    */
/**********************************************************************/

#include        "header.h"
#include        <sys/time.h>
#include        <sys/resource.h>


#define MICROSEC        0.0625  /* 1/(16MHz) in m-sec                 */


extern  char    d_a[MAXDATA];           /* list a for binary operations          */
extern  char    d_b[MAXDATA];           /* list b                                */
extern  char    dc_a[MAXDATA];          /* copy of list a for binary operations  */
extern  char    dc_b[MAXDATA];          /* copy of list b                        */
extern  int     useMask;                /* true iff masks are used              */
extern  char    mask[512];              /* contains mask bytes if useMask       */
extern  int     k;                      /* # bytes in the key                    */
extern  int     kp;                     /* # bytes in each record                */
extern  char    filea_name[10];
extern  char    fileb_name[10];
extern  long    dataSize_a;             /* # bytes in list a                     */
extern  long    dataSize_b;             /* # bytes in list b                     */
extern  long    binarySearch();
extern  long    binarySearch2();
struct  rusage          startTime;
struct  rusage          endTime;
char    result[MAXDATA]; /* result of intersection */


intersection()
{

        float   time;
        FILE    *fopen();
        FILE    *fpout, *fpstat;
        register long    i;
        long    copyCount;
        long    copyTime;
        long    sortTime;
        long    intCount;
        long    intTime;
        long    sortCount;
        long    temp1,temp2,temp3;
        long    dataSize;
        long    milliseconds;
        register char    *d;
        register char    *ptr;
        register char    *addrptr;
        register char    *ptr1;
        char    *list;
        register char   *skey;
        char    *strncpy();
        register int    noFind;
        register int     done;
        char    *upper;
        char    *lower;
        char    *malloc();
        register int    numrecs;
```

```
sortCount = 0;
copyCount = 0;
intCount = 0;
intTime = 0;


ptr = dc_a;
skey = malloc(k);
upper = result;

/* time to copy */
getrusage( RUSAGE_SELF, &startTime);
endTime.ru_utime = startTime.ru_utime;
while( endTime.ru_utime.tv_sec - startTime.ru_utime.tv_sec < 30 ) {
        for( i=0; i < dataSize_a; i++) {
        dc_a[i] = d_a[i];
}
        for( i=0; i < dataSize_b; i++) {
        dc_b[i] = d_b[i];
}
strncpy(skey,ptr,k);
copyCount++;
getrusage( RUSAGE_SELF, &endTime);
}
copyTime = 1000000 *
( endTime.ru_utime.tv_sec - startTime.ru_utime.tv_sec ) +
( endTime.ru_utime.tv_usec - startTime.ru_utime.tv_usec );
copyTime += 1000000 *   /* system */
( endTime.ru_stime.tv_sec - startTime.ru_stime.tv_sec ) +
( endTime.ru_stime.tv_usec - startTime.ru_stime.tv_usec );


getrusage( RUSAGE_SELF, &startTime);
endTime.ru_utime = startTime.ru_utime;
while( endTime.ru_utime.tv_sec - startTime.ru_utime.tv_sec < 30 ) {
        for( i=0; i < dataSize_a; i++) {
        dc_a[i] = d_a[i];
        }
        for( i=0; i < dataSize_b; i++) {
        dc_b[i] = d_b[i];
        }
        trosq(d_a,(int)dataSize_a/kp,kp,keyComp);
        trosq(d_b,(int)dataSize_b/kp,kp,keyComp);
sortCount++;
getrusage( RUSAGE_SELF, &endTime);
}
sortTime = 1000000 *
( endTime.ru_utime.tv_sec - startTime.ru_utime.tv_sec ) +
( endTime.ru_utime.tv_usec - startTime.ru_utime.tv_usec );
sortTime += 1000000 *   /* system */
( endTime.ru_stime.tv_sec - startTime.ru_stime.tv_sec ) +
( endTime.ru_stime.tv_usec - startTime.ru_stime.tv_usec );


getrusage( RUSAGE_SELF, &startTime);
endTime.ru_utime = startTime.ru_utime;
while( endTime.ru_utime.tv_sec - startTime.ru_utime.tv_sec < 30 ) {

if(dataSize_b / kp > dataSize_a / kp)
  {
    d = d_b;
    list  = d_a;
    dataSize = dataSize_b;
    numrecs = dataSize_a / kp;
```

```
          }
        else
          {
            d = d_a;
            list = d_b;
            dataSize = dataSize_a;
            numrecs = dataSize_b / kp;
          }
/* now, d points to larger list, dataSize is its size,  */
/*list points to smaller list  with numrecs records.    */


        ptr = list;
        ptr1 = upper;
        done = 0;


        while(numrecs--  && !done)
         {
            strncpy(skey,ptr,k);
            addrptr = d + binarySearch2(skey,d,dataSize);
            noFind = keyComp(skey, addrptr);
            done = (addrptr >= (d + dataSize));
            if(!noFind)
               for(i=0;i<kp;i++) *ptr1++ = *ptr++;
         }


       intCount++;
       getrusage( RUSAGE_SELF, &endTime);
   }  /* end while */
       temp1 = 1000000 *
       ( endTime.ru_utime.tv_sec - startTime.ru_utime.tv_sec ) +
       ( endTime.ru_utime.tv_usec - startTime.ru_utime.tv_usec );
       temp1 += 1000000 *      /* system */
       ( endTime.ru_stime.tv_sec - startTime.ru_stime.tv_sec ) +
       ( endTime.ru_stime.tv_usec - startTime.ru_stime.tv_usec );
       temp1 = temp1 / intCount;
       temp2 = copyTime / copyCount;   /* time for copy and sys call*/
       temp3 = sortTime / sortCount;   /* time for sort*/
       milliseconds = (temp1  + temp3 - temp2) / 1000;


       fpout = fopen("result.out","w");        /* output file */
       fpstat = fopen("result.stat","w");      /* statistics */

       result[upper - ptr1] = '\0';
       fprintf(fpout,"%s",result);

       fprintf(fpstat,"\n\n\t\tSORT ENGINE  SIMULATOR  STATISTICS\n\n");
       fprintf(fpstat,"\t\t\t\tSUN 3/50\n\n");
       fprintf(fpstat,"\t\tOperation :\t\tIntersection\n\n");
       fprintf(fpstat,"\n\t\tInput File :\t\t%s\n",filea_name);
       fprintf(fpstat,"\n\t\tRecords File :\t\t%s\n",fileb_name);
       fprintf(fpstat,"\n\t\t# Records_a :\t\t%d\n",dataSize_a / kp);
       fprintf(fpstat,"\n\t\t# Records_b :\t\t%d\n",dataSize_b / kp);
       fprintf(fpstat,"\n\t\tKey length : \t\t%d\n",k);
       fprintf(fpstat,"\n\t\tPointer length : \t%d\n",kp-k);
       fprintf(fpstat,"\n\t\tSort time : \t\t%ld millisecs\n",(temp3 - temp2)  / 1000);
       fprintf(fpstat,"\n\t\tIntersection time : \t%ld millisecs\n",temp1 /  1000);
       fprintf(fpstat,"\n\t\tTotal time : \t\t%ld millisecs\n",milliseconds) ;
       fprintf(fpstat,"\n\n\t\t\t*********************\n");
       fclose(fpout);
       fclose(fpstat);

 }  /* end of main */
```

```
/**********************************************************************/
/*                                                                    */
/* sw_select.c -- calculates the time necessary to find all occurences  */
/* of a given record.   i.e., returns time required to perform selection*/
/* on a SUN 3/50                                                      */
/*                                                                    */
/*                                                                    */
/* January  9, 1987                                                   */
/*                                                                    */
/**********************************************************************/

#include        "header.h"
#include        <sys/time.h>
#include        <sys/resource.h>


#define MICROSEC        0.0625  /* 1/(16MHz) in m-sec                  */


extern  char    d_a[MAXDATA];           /* list a for binary operations         */
extern  char    d_b[MAXDATA];           /* list b                               */
extern  char    dc_a[MAXDATA];          /* copy of list a for binary operations */
extern  char    dc_b[MAXDATA];          /* copy of list b                       */
extern  int     useMask;                /* true iff masks are used             */
extern  char    mask[512];              /* contains mask bytes if useMask      */
extern  int     k;                      /* # bytes in the key                   */
extern  int     kp;                     /* # bytes in each record               */
extern  char    filea_name[10];
extern  char    fileb_name[10];
extern  long    dataSize_a;             /* # bytes in list a                    */
extern  long    dataSize_b;             /* # bytes in list b                    */
extern  long    binarySearch();
extern  long    binarySearch2();
struct  rusage          startTime;
struct  rusage          endTime;


select()
{
        extern  char    pred;           /* for selection
                                                =  ==  a
                                                >  ==  b
                                                >= ==  c
                                                <  ==  d
                                                <= ==  e
                                          range    ==  f  */

        float   time;
        FILE    *fopen();
        FILE    *fpout, *fpstat;
        long    i;
        long    copyCount;
        long    copyTime;
        long    sortCount;
        long    sysCallTime;    /* time for a system call to check time */
        long    temp1,temp2;
        long    dataSize;
        long    milliseconds;
        char    *d;
        char    *ptr;
        char    *list;
        char    *skey;
        char    *strncpy();
        int     nofind;
        char    *upper;
        char    *lower;
```

```
long    x,y;
char    *malloc();
register        int     numrecs;        /* # records to be found (selected ) */
register        int     count;
register        int     z;


sortCount = 0;
copyCount = 0;
dataSize = dataSize_b;
d = dc_a;
skey = malloc(k);
upper = d_b;
lower = dc_b;
x=0;
while(x<dataSize)
{
 for(y=0;y<k;y++)
  {
    *lower++ = *upper++;
    x++;
  }
  upper++;      /* get rid of \n */
  x++;
}

dataSize_b = lower - dc_b;
numrecs = dataSize_b / k;


 if((dataSize_b % k )!= 0)
 {
   fprintf(stderr,"\nYour key is not %d bytes long\n",k);
   exit(-1);
 }

dataSize = dataSize_a;

/* time to copy */
getrusage( RUSAGE_SELF, &startTime);
endTime.ru_utime = startTime.ru_utime;
while( endTime.ru_utime.tv_sec - startTime.ru_utime.tv_sec < 30 ) {
        for( i=0; i < dataSize; i++) {
        dc_a[i] = d_a[i];
}
strncpy(skey, d_b, k);
copyCount++;
getrusage( RUSAGE_SELF, &endTime);
}
copyTime = 1000000 *
( endTime.ru_utime.tv_sec - startTime.ru_utime.tv_sec ) +
( endTime.ru_utime.tv_usec - startTime.ru_utime.tv_usec );
copyTime += 1000000 *   /* system */
( endTime.ru_stime.tv_sec - startTime.ru_stime.tv_sec ) +
( endTime.ru_stime.tv_usec - startTime.ru_stime.tv_usec );


getrusage( RUSAGE_SELF, &startTime);
endTime.ru_utime = startTime.ru_utime;
while( endTime.ru_utime.tv_sec - startTime.ru_utime.tv_sec < 30 ). {
        for( i=0; i < dataSize; i++) {
                d[i] = d_a[i];
        }
        trosq(d,(int)dataSize/kp,kp,keyComp);

switch(pred)
```

```
{
    case 'a' :          /*  ==   */
    ptr = d_b;
    lower = dc_b;
    for(count=0; count < numrecs; count++)
    {
       strncpy(skey, lower, k);
       lower += k;
       upper =  d + binarySearch2(skey,d,dataSize);
       nofind = keyComp( skey, upper);
       while (! nofind )
       {
          for(z=0;z<kp;z++) *ptr++ = *upper++;
          nofind = keyComp( skey, upper);
        }
     }
     *ptr = '\0';
    break;

    case 'b' :          /*  >  */
        lower = dc_b;
        strncpy(skey, lower, k);
        upper =  d + binarySearch(skey,d,dataSize);
        numrecs = (d + dataSize) - upper;
        for(count = 0; count < numrecs; count++)
          d_b[count] = *upper++;
        d_b[count] = '\0';

    break;

    case 'c' :          /*  >=  */
        lower = dc_b;
        strncpy(skey, lower, k);
        upper =  d + binarySearch2(skey,d,dataSize);
        numrecs = (d + dataSize) - upper;
        for(count = 0; count < numrecs; count++)
          d_b[count] = *upper++;
        d_b[count] = '\0';

    break;

    case 'd' :          /* < */
        lower = dc_b;
        strncpy(skey, lower, k);
        upper =  d + binarySearch2(skey,d,dataSize);
        numrecs = upper - d;
        upper = d;
        for(count = 0; count < numrecs; count++)
          d_b[count] = *upper++;
        d_b[count] = '\0';

    break;

    case 'e' :          /* <= */
        lower = dc_b;
        strncpy(skey, lower, k);
        upper =  d + binarySearch(skey,d,dataSize);
        numrecs = upper - d;
        upper = d;
        for(count = 0; count < numrecs; count++)
          d_b[count] = *upper++;
        d_b[count] = '\0';

    break;

    case 'f' :          /* <> i.e., > 1st key, < 2nd key*/
```

```
            lower = dc_b;
            strncpy(skey, lower, k);
            lower =  d + binarySearch(skey,d,dataSize);
            upper = dc_b + k;
            strncpy(skey, upper, k);
            upper =  d + binarySearch2(skey,d,dataSize);
            numrecs = upper - lower;
            for(count = 0; count < numrecs; count++)
              d_b[count] = *lower++;
            d_b[count] = '\0';

        }


        sortCount++;
        getrusage( RUSAGE_SELF, &endTime);
    }  /* end while */
        temp1 = 1000000 *
        ( endTime.ru_utime.tv_sec - startTime.ru_utime.tv_sec ) +
        ( endTime.ru_utime.tv_usec - startTime.ru_utime.tv_usec );
        temp1 += 1000000 *      /* system */
        ( endTime.ru_stime.tv_sec - startTime.ru_stime.tv_sec ) +
        ( endTime.ru_stime.tv_usec - startTime.ru_stime.tv_usec );
        temp1 = temp1 / sortCount;
        temp2 = copyTime / copyCount;   /* time for copy and sys call*/
        milliseconds = (temp1  - temp2) / 1000;


        fpout = fopen("result.out","w");         /* output file */
        fpstat = fopen("result.stat","w");       /* statistics */

        fprintf(fpout,"%s",d_b);

        fprintf(fpstat,"\n\n\t\tSORT ENGINE  SIMULATOR  STATISTICS\n\n");
        fprintf(fpstat,"\t\t\t\tSUN 3/50\n\n");
        fprintf(fpstat,"\t\tOperation :\t\tSelection\n\n");
        fprintf(fpstat,"\n\t\tInput File :\t\t%s\n",filea_name);
        fprintf(fpstat,"\n\t\tRecords File :\t\t%s\n",fileb_name);
        fprintf(fpstat,"\n\t\t# Records :\t\t%d\n",dataSize_a / kp);
        fprintf(fpstat,"\n\t\tKey length : \t\t%d\n",k);
        fprintf(fpstat,"\n\t\tPointer length : \t%d\n",kp-k);
        fprintf(fpstat,"\n\t\tTotal time : \t\t%ld millisecs\n",milliseconds);
        fprintf(fpstat,"\n\n\t\t\t*********************\n");
        fclose(fpout);
        fclose(fpstat);

}  /* end of main */
```

```
/************************************************************************/
/*                                                                      */
/* sw_intersection.c -- calculates the time necessary to find intersection      */
/* of a given list with another list on a SUN 3/50                      */
/*                                                                      */
/*                                                                      */
/* January  9, 1987                                                     */
/*                                                                      */
/************************************************************************/

#include        "header.h"
#include        <sys/time.h>
#include        <sys/resource.h>


#define MICROSEC        0.0625  /* 1/(16MHz) in m-sec                    */


extern  char    d_a[MAXDATA];           /* list a for binary operations         */
extern  char    d_b[MAXDATA];           /* list b                               */
extern  char    dc_a[MAXDATA];          /* copy of list a for binary operations  */
extern  char    dc_b[MAXDATA];          /* copy of list b                       */
extern  int     useMask;                /* true iff masks are used             */
extern  char    mask[512];              /* contains mask bytes if useMask      */
extern  int     k;                      /* # bytes in the key                   */
extern  int     kp;                     /* # bytes in each record               */
extern  char    filea_name[10];
extern  char    fileb_name[10];
extern  long    dataSize_a;             /* # bytes in list a                    */
extern  long    dataSize_b;             /* # bytes in list b                    */
extern  long    binarySearch();
extern  long    binarySearch2();
struct  rusage          startTime;
struct  rusage          endTime;
char    result[MAXDATA]; /* result of intersection */


intersection()
{

        float   time;
        FILE    *fopen();
        FILE    *fpout, *fpstat;
        register long    i;
        long    copyCount;
        long    copyTime;
        long    sortTime;
        long    intCount;
        long    intTime;
        long    sortCount;
        long    temp1,temp2,temp3;
        long    dataSize;
        long    milliseconds;
        register char    *d;
        register char    *ptr;
        register char    *addrptr;
        register char    *ptr1;
        char    *list;
        register char   *skey;
        char    *strncpy();
        register int    noFind;
        register int     done;
        char    *upper;
        char    *lower;
        char    *malloc();
        register int    numrecs;
```

```
sortCount = 0;
copyCount = 0;
intCount = 0;
intTime = 0;


ptr = dc_a;
skey = malloc(k);
upper = result;

/* time to copy */
getrusage( RUSAGE_SELF, &startTime);
endTime.ru_utime = startTime.ru_utime;
while( endTime.ru_utime.tv_sec - startTime.ru_utime.tv_sec < 30 ) {
        for( i=0; i < dataSize_a; i++) {
        dc_a[i] = d_a[i];
}
        for( i=0; i < dataSize_b; i++) {
        dc_b[i] = d_b[i];
}
strncpy(skey,ptr,k);
copyCount++;
getrusage( RUSAGE_SELF, &endTime);
}
copyTime = 1000000 *
( endTime.ru_utime.tv_sec - startTime.ru_utime.tv_sec ) +
( endTime.ru_utime.tv_usec - startTime.ru_utime.tv_usec );
copyTime += 1000000 *   /* system */
( endTime.ru_stime.tv_sec - startTime.ru_stime.tv_sec ) +
( endTime.ru_stime.tv_usec - startTime.ru_stime.tv_usec );


getrusage( RUSAGE_SELF, &startTime);
endTime.ru_utime = startTime.ru_utime;
while( endTime.ru_utime.tv_sec - startTime.ru_utime.tv_sec < 30 ) {
        for( i=0; i < dataSize_a; i++) {
        dc_a[i] = d_a[i];
        }
        for( i=0; i < dataSize_b; i++) {
        dc_b[i] = d_b[i];
        }
        trosq(d_a,(int)dataSize_a/kp,kp,keyComp);
        trosq(d_b,(int)dataSize_b/kp,kp,keyComp);
sortCount++;
getrusage( RUSAGE_SELF, &endTime);
}
sortTime = 1000000 *
( endTime.ru_utime.tv_sec - startTime.ru_utime.tv_sec ) +
( endTime.ru_utime.tv_usec - startTime.ru_utime.tv_usec );
sortTime += 1000000 *   /* system */
( endTime.ru_stime.tv_sec - startTime.ru_stime.tv_sec ) +
( endTime.ru_stime.tv_usec - startTime.ru_stime.tv_usec );


getrusage( RUSAGE_SELF, &startTime);
endTime.ru_utime = startTime.ru_utime;
while( endTime.ru_utime.tv_sec - startTime.ru_utime.tv_sec < 30 ) {

if(dataSize_b / kp > dataSize_a / kp)
  {
    d = d_b;
    list  = d_a;
    dataSize = dataSize_b;
    numrecs = dataSize_a / kp;
```

```
            }
         else
            {
              d = d_a;
              list = d_b;
              dataSize = dataSize_a;
              numrecs = dataSize_b / kp;
            }
/* now, d points to larger list, dataSize is its size,  */
/*list points to smaller list  with numrecs records.    */


         ptr = list;
         ptr1 = upper;
         done = 0;


         while(numrecs--  && !done)
          {
             strncpy(skey,ptr,k);
             addrptr = d + binarySearch2(skey,d,dataSize);
             noFind = keyComp(skey, addrptr);
             done = (addrptr >= (d + dataSize));
             if(!noFind)
                for(i=0;i<kp;i++) *ptr1++ = *ptr++;
          }


        intCount++;
        getrusage( RUSAGE_SELF, &endTime);
   }  /* end while */
        temp1 = 1000000 *
        ( endTime.ru_utime.tv_sec - startTime.ru_utime.tv_sec ) +
        ( endTime.ru_utime.tv_usec - startTime.ru_utime.tv_usec );
        temp1 += 1000000 *      /* system */
        ( endTime.ru_stime.tv_sec - startTime.ru_stime.tv_sec ) +
        ( endTime.ru_stime.tv_usec - startTime.ru_stime.tv_usec );
        temp1 = temp1 / intCount;
        temp2 = copyTime / copyCount;   /* time for copy and sys call*/
        temp3 = sortTime / sortCount;   /* time for sort*/
        milliseconds = (temp1  + temp3 - temp2) / 1000;


        fpout = fopen("result.out","w");        /* output file */
        fpstat = fopen("result.stat","w");      /* statistics */

        result[upper - ptr1] = '\0';
        fprintf(fpout,"%s",result);

        fprintf(fpstat,"\n\n\t\tSORT ENGINE  SIMULATOR  STATISTICS\n\n");
        fprintf(fpstat,"\t\t\t\tSUN 3/50\n\n");
        fprintf(fpstat,"\t\tOperation :\t\tIntersection\n\n");
        fprintf(fpstat,"\n\t\tInput File :\t\t%s\n",filea_name);
        fprintf(fpstat,"\n\t\tRecords File :\t\t%s\n",fileb_name);
        fprintf(fpstat,"\n\t\t# Records_a :\t\t%d\n",dataSize_a / kp);
        fprintf(fpstat,"\n\t\t# Records_b :\t\t%d\n",dataSize_b / kp);
        fprintf(fpstat,"\n\t\tKey length : \t\t%d\n",k);
        fprintf(fpstat,"\n\t\tPointer length : \t%d\n",kp-k);
        fprintf(fpstat,"\n\t\tSort time : \t\t%ld millisecs\n",(temp3 - temp2) / 1000);
        fprintf(fpstat,"\n\t\tIntersection time : \t%ld millisecs\n",temp1 / 1000);
        fprintf(fpstat,"\n\t\tTotal time : \t\t%ld millisecs\n",milliseconds);
        fprintf(fpstat,"\n\n\t\t\t*********************\n");
        fclose(fpout);
        fclose(fpstat);

}  /* end of main */
```

```
/*****************************************************************************/
/*                                                                           */
/* This function calculates the timing for sorting the input using Brian's   */
/* implementation of quicksort.                                              */
/*                                                                           */
/*****************************************************************************/

#include "header.h"
#include        <sys/time.h>
#include        <sys/resource.h>


extern  int     useMask;                /* true iff masks are used          */
extern  char    mask[512];              /* contains mask bytes if useMask   */
extern  char    d_a[MAXDATA];           /* list a for binary operations      */
extern  char    d_b[MAXDATA];           /* list b                            */
extern  char    dc_a[MAXDATA];          /* copy of list a for binary operations */
extern  char    dc_b[MAXDATA];          /* copy of list b                    */
extern  int     k;                      /* # bytes in the key                */
extern  int     kp;                     /* # bytes in each record            */
extern  char    filea_name[10];
extern  char    fileb_name[10];
extern  long    dataSize_a;             /* # bytes in list a                 */
extern  long    dataSize_b;             /* # bytes in list b                 */
extern  long    dataSize;
struct  rusage          startTime;
struct  rusage          endTime;



quicksort()
 {

        float   time;
        FILE    *fopen();
        FILE    *fpout, *fpstat;
        long    i;
        long    copyCount;
        long    copyTime;
        long    sortCount;
        long    sysCallTime;    /* time for a system call to check time */
        long    temp1,temp2;
        long    milliseconds;
        char    *d;
        /* time the sort itself */
        sortCount = 0;
        copyCount = 0;
        dataSize = dataSize_a;

/* time to copy */
        getrusage( RUSAGE_SELF, &startTime);
        endTime.ru_utime = startTime.ru_utime;
        while( endTime.ru_utime.tv_sec - startTime.ru_utime.tv_sec < 30 ) {
        for( i=0; i < dataSize; i++) {
                dc_a[i] = d_a[i];
        }
        copyCount++;
        getrusage( RUSAGE_SELF, &endTime);
        }
        copyTime = 1000000 *
        ( endTime.ru_utime.tv_sec - startTime.ru_utime.tv_sec ) +
        ( endTime.ru_utime.tv_usec - startTime.ru_utime.tv_usec );
        copyTime += 1000000 *   /* system */
        ( endTime.ru_stime.tv_sec - startTime.ru_stime.tv_sec ) +
        ( endTime.ru_stime.tv_usec - startTime.ru_stime.tv_usec );
        d = dc_a;
        getrusage( RUSAGE_SELF, &startTime);
        endTime.ru_utime = startTime.ru_utime;
        while( endTime.ru_utime.tv_sec - startTime.ru_utime.tv_sec < 30 ) {
                for( i=0; i < dataSize; i++) {
                        d[i] = d_a[i];
                }
            trosq(d,(int)dataSize/kp,kp,keyComp);
            sortCount++;
            getrusage( RUSAGE_SELF, &endTime);
        }  /* end while */
        temp1 = 1000000 *
        ( endTime.ru_utime.tv_sec - startTime.ru_utime.tv_sec ) +
        ( endTime.ru_utime.tv_usec - startTime.ru_utime.tv_usec );
        temp1 += 1000000 *       /* system */
        ( endTime.ru_stime.tv_sec - startTime.ru_stime.tv_sec ) +
        ( endTime.ru_stime.tv_usec - startTime.ru_stime.tv_usec );
        temp1 = temp1 / sortCount;
        temp2 = copyTime / copyCount;   /* time for copy and sys call*/
        milliseconds = (temp1  - temp2) / 1000;

        fpout = fopen("result.out","w");        /* output file */
        fpstat = fopen("result.stat","w");      /* statistics */
        d[dataSize] = '\0';
        fprintf(fpout,"%s",d);

        fprintf(fpstat,"\n\n\t\tSORT ENGINE  SIMULATOR  STATISTICS\n\n");
        fprintf(fpstat,"\t\t\t\tSUN 3/50\n\n");
        fprintf(fpstat,"\t\tOperation :\t\tSorting\n\n");
        fprintf(fpstat,"\n\t\tFile :\t\t\t%s\n",filea_name);
        fprintf(fpstat,"\n\t\t# Records :\t\t%d\n",dataSize_a / kp);
        fprintf(fpstat,"\n\t\tKey length : \t\t%d\n",k);
        fprintf(fpstat,"\n\t\tPointer length : \t%d\n",kp-k);
        fprintf(fpstat,"\n\t\tTotal time : \t\t%ld millisecs\n",milliseconds);
        fprintf(fpstat,"\n\n\t\t\t*********************\n");
        fclose(fpout);
        fclose(fpstat);
}
```

BIBLIOGRAPHY

[AMD 86] "Am95C85 (CADM) Technical Manual" *Advanced Micro Devices*, Austin, TX., 1986.

[AMD2 86] "CADM Sortmerge Performance",*Advanced Micro Devices*, Austin, TX., 1986.

[BHAT 86] Bhat, V. and Anderson, A., "CADM Based Search Utility for TRS-80 Computer", *Departmennt of Elec. Engg., Project report for EE 380N*, The University of Texas at Austin, TX., December 1986.

[BROW 85] Browne, J.C., A.Dale, C.Leung and R.Jenevein, "A Parallel Multi-Stage I/O Architecture with a Self-Managing Disk Cache for Database Management Applications", *Proceedings of the Fourth International Workshop on Database Machines*, March 1985, pp.330-345.

[HELL 81] Hell, W., et al, "RDBM : A Dedicated Multiprocessor System for Database Management", Chapter 3 in *Advanced Database Machine Architecture*, Prentice Hall, 1981.

[JENE 86] Jenevein,R., A.Dale, B.Menezes and K.Thadani, "Design of a HyperKYKLOS-based Multiprocessor Architecture for High-Performance Join Operations", *Dept. of Computer Sciences, Technical Report TR-87-18*, The University of Texas at Austin, Austin, Tx., May 1987.

[KNUT 73] Knuth, D.E., "Sorting and Searching", *The Art of Computer Programming*, pp 160, Vol. 3, Addison-Wessley 1973.

[LIPO 75] Lipovski, G.J., and Su, S.Y.W., "CASSM : A Cellular System for Very Large Databases", *Proceedings of the Conference on Very Large Databases*, September 1975.

[MENE 85] Menezes,B., and R.Jenevein, "KYKLOS : A Linear growth Fault-tolerant Interconnection Network", *Proceedings of the International Conference on Parallel Processing*, pp.498-502, August 1985.

[MOTO 87] "16/32 Bit Microcomoputer System Components", *Motorola Inc.*, pp. 1.1 to 1.26, Phoenix, AZ., 1987.

[OZKA 75] Ozkarahan, D.A., Schuster, S.A., Nguyen, H.B. and Smith, K.C., "RAP.2 : An Associative Processor for Databases and its

Applications", *IEEE Transactions on Computers*, Vol. C28, No.6, June 1979.

[SCHW 83] Schweppe, H., Zeidler, H.C., Hell, W., Leilich, H.O., Stieg, G., Teich, W., "RDBM : A Dedicated Multiprocessor System for Database Manegement", *Chapter 3 in Hsiao, D., "Advanced Database Machine Architecture"*, Prentice-Hall, 1983.

[SAKA 84] Sakai, Hiroshi., et. al., "Design and Implementation of the Relational Database Engine", *Proceedings of the International Conference on Fifth Generation Computer Systems*, 1984.