μDatabase : A Toolkit for Constructing Memory Mapped Databases

Peter A. Buhr, Anil K. Goel and Anderson Wai
Dept. of Computer Science, University of Waterloo
Waterloo, Ontario, Canada, N2L 3G1

Abstract

The main objective of this work was an efficient methodology for constructing low-level database tools that are built around a single-level store implemented using memory mapping. The methodology allowed normal programming pointers to be stored directly onto secondary storage, and subsequently retrieved and manipulated by other programs without the need for relocation, pointer swizzling or reading all the data. File structures for a database, e.g. a B-Tree, built using this approach are significantly simpler to build, test, and maintain than traditional file structures. All access methods to the file structure are statically type-safe and file structure definitions can be generic in the type of the record and possibly key(s) stored in the file structure, which affords significant code reuse. An additional design requirement is that multiple file structures may be simultaneously accessible by an application. Concurrency at both the front end (multiple accessors) and the back end (file structure partitioned over multiple disks) are possible. Finally, experimental results between traditional and memory mapped files structures show that performance of a memory mapped file structure is as good or better than the traditional approach.

1 Introduction

The main objective of this work was an efficient methodology for constructing low-level database tools that are built around a single-level store. A *single-level store* gives the illusion that data on disk (*secondary storage*) is accessible in the same way as data in main memory (*primary storage*), which is analogous to the goals of virtual memory. This uniform view of data eliminates the need for complex and expensive execution-time conversions of structured data between primary and secondary storage. A uniform view of data also allows the expressive power and the data structuring capabilities of a general purpose programming language to be used in creating and manipulating data structures stored on secondary storage. Although a single-level store is an old idea [Org72, IBM78], it has seen only limited use inside of operating systems, and it is only during the last few years that this idea has begun to receive new attention and approval from researchers in the database and programming language communities [CFW90, SZ90a, LLOW91]. For complex structures, a single-level store offers substantial performance advantages over conventional file access, which is crucial to database applications such as CAD/CAM systems, text management and GIS [vO90]. We argue that the performance advantage of a single-level store is lost if the pointers within it have to be relocated or swizzled [CAC+84, Mos90, Wil91].

One way of efficiently implementing a single-level store is by means of memory mapped files. Memory mapping is the use of virtual memory to map files stored on secondary storage into primary storage so that the data is directly accessible by the processor's instructions. Therefore, explicit read and write routine calls are not used to access data on disk. All read and write operations are done implicitly by the operating system during execution of a program. When the working set of the data structure can be kept in memory, performance begins to approach that of memory-resident databases.

To show the efficiency of memory mapping, a memory mapped implementation was constructed, which allowed file access experiments to be performed between traditional and memory mapped schemes. A tool kit approach was adopted for the implementation because it allows programmers to participate in some of the design activity; the tool kit is called μDatabase. Persistence in μDatabase is orthogonal because creating and manipulating data structures in a persistent area is the same as in a program. μDatabase is intented to provide easy-to-use and efficient tools for developing new databases, and for maintaining existing databases. While μDatabase shares the underlying principles of a single-level store with other recent proposals [CFW90, SZ90a, LLOW91, STP+87], it offers features that make it unique and an attractive alternative. μDatabase is *not* an object store but it could be used to implement one.

In this paper, a *file structure* is defined to be a data structure that is a container for user records on secondary storage; a file structure relates the records in a particular way, for example, maintaining the records in order by one or more keys. An *access method* is defined to be a particular way that records are accessed. Examples of different access methods are: initial loading of records, sequential access of records, keyed access of records.

In Antonio Albano and Ron Morrison, editors, *Persistent Object Systems*, pages 166–185, San Miniato, Italy, September 1992. Springer-Verlag. Workshops in Computing, Ed. by Professor C. J. van Rijsbergen, QA76.9.D3I59.

2 Motivation

A database programmer is faced with the problem of dealing with two different views of structured data, viz. the data in primary storage and the data on secondary storage. Traditionally, these two views of data tend to be incompatible with each other. It is extremely difficult and cumbersome to construct complex relationships among different objects without the help of direct pointers. However, it is generally impossible to store and retrieve data structures containing pointers from disk without converting at least the pointers and at worst the entire data structure into a different format. Considerable efforts, both in terms of programming and execution time, have to be made in such systems to transform data from one view to the other. In general, these transformations are data structure specific and must be executed each time the data structure is stored or read from secondary storage. Furthermore, the powerful and flexible data structuring capabilities of modern programming languages are not directly available for building data structures on secondary storage.

In spite of these rather taxing difficulties, database implementors have traditionally rejected the use of mapped files and have chosen to implement the lower-level support for databases themselves using traditional approaches. This rejection is not totally based on the lack of memory mapping facilities. The earliest use of memory mapping techniques can be traced back 20 years to the Multics system. However Multics provided these facilities in a framework that was very rigid and difficult to work with. More recent operating systems have begun to provide means for implementing the idea of a single-level store. See [SZ90a, p. 90] for other reasons why mapped files have not been popular with database designers. All of these reasons are now addressed by new operating systems [TRY$^+$87, Sun90], which provide extended access to the virtual memory, and new hardware, which provides large address spaces (64 bits) and N-level paging [Mip91, RKA92].

3 Memory Mapping

3.1 Disadvantages of Memory Mapping

Larger Pointers Memory pointers may be larger than disk offsets, which increases the size of the file structure marginally increasing access cost.

Non-Uniform Access Speed The apparent direct access of data can give a false sense of control to the file structure designer. While a file's contents are directly accessible to the processor, the access speed is non-uniform—when a non-resident page is referenced, a long delay occurs as for a traditional I/O operation; otherwise the reference is direct and occurs at normal memory speed. When programming a file structure using memory mapping, certain data structures will be inappropriate because of their access patterns.

3.2 Advantages of Memory Mapping

Common Data Structure in Primary and Secondary Memory Use of programming-language data structures to organize the contents of a file eliminates the need to convert to a secondary storage format, which results in code that is substantially more reliable and easier to maintain. Also, for complex data structures, like an object in a CAD/CAM system, there is a significant performance advantage.

Reduced Need for Explicit Buffer Management A sophisticated buffer manager is crucial for the performance of a traditional database system. Furthermore, a file structure designer must be skilled in its use, explicitly invoking its facilities and pinning/unpinning buffers. On systems without pinning support, double paging is a serious drawback. A memory mapped access method is less complex because I/O management is largely transparent and is handled at the lowest possible level (instruction fetch and store).

Simple Localization While locality of references is crucial for all data structures where access is non-uniform, memory-mapped access methods can easily take advantage of it by controlling memory layout. Because the data structures on secondary storage can be manipulated directly by the programming language, tuning for localization is straightforward.

Rapid Prototyping of Access Methods Because a file structure designer works with a uniform view of data, a file structure can be reliably constructed in a short period of time, using all the available programming-language tools. Polymorphism, interactive debuggers, execution and storage profilers, and visualization tools are some examples of directly usable aids.

Memory Mapping on a Loaded System It is our contention that memory-mapped access methods can potentially achieve better performance than traditional database systems, particularly on a shared system. A buffer manager is often in conflict with other applications, in particular, holding storage that it is not using. On the other hand, memory-mapped access methods can immediately take advantage of available storage to reduce I/O operations.

Contiguous Address Space Memory mapping provides the file structure designer with a contiguous address space even when the data on secondary storage is not contiguous. A single object within a given file structure may be split into several extents on one disk or across multiple disks, and a file structure designer may see nothing difference or only a sparse address space.

4 μDatabase Design Methodology

Instead of using reachability [PS-87, MBC+89], μDatabase uses the notion of a persistent area, in which data objects can be built or copied if they are to persist [BZ86, BZ89]. A persistent area is currently implemented by an operating system file. If data is to be transferred from one persistent area to another, the data must be copied through an intermediate area. Alternatively, persistent data can be manipulated directly by migrating an application task to the persistent area and perform the operations directly on the data. The amount of task migration and/or copying depends on the size of the data and the amount of work performed when manipulating the data. In all cases, the user interface to the file structure provides encapsulation to ensure its integrity.

An application may need to access several persistent areas simultaneously. Our design requirements mandate that support for multiple accessible file structures in a single application be provided, while allowing each file structure to use conventional pointers without having to adjust them. To accomplish this requirement, each persistent area is mapped into its own segment. This approach is in contrast to systems that provide simultaneous access by mapping multiple persistent areas into the same segment. In these systems, all pointers are relocated when portions of an area are mapped. In general, this requires access to the type information of the file structure at runtime, which is not usually possible in programming languages that do not have runtime type-checking. Also, significant execution overhead is incurred in relocating pointers.

Currently, μDatabase does *not* cover pointers among persistent areas (see [BZ89] for a possible solution). Nor does it deal with distributed persistent areas; we believe that distributed shared memory [SZ90b, WF90] will allow our current design to scale up to a distributed environment. Object-oriented programming techniques are employed in the implementation of μDatabase, but are not essential. μC++ [BDS+92] is used as the implementation language, which is a superset of C++ with concurrency extensions, because it allows immediate technology transfer.

The following two properties evolved during the design and implementation of μDatabase. First, data associated with accessing a file structure, such as current location in the file, concurrency data or transient recovery information are not mapped in the file structure. Second, a deliberate attempt is made to retain the conventional semantics of *opening* and *closing* a file. Implicit schemes, like pointer-swizzling, have problems detecting the first access but the most difficult problem is knowing when the access can be terminated (garbage collectors are too slow). The properties involve several levels, each performing a particular aspect of the storage or access management of the file structure (see Figure 1).

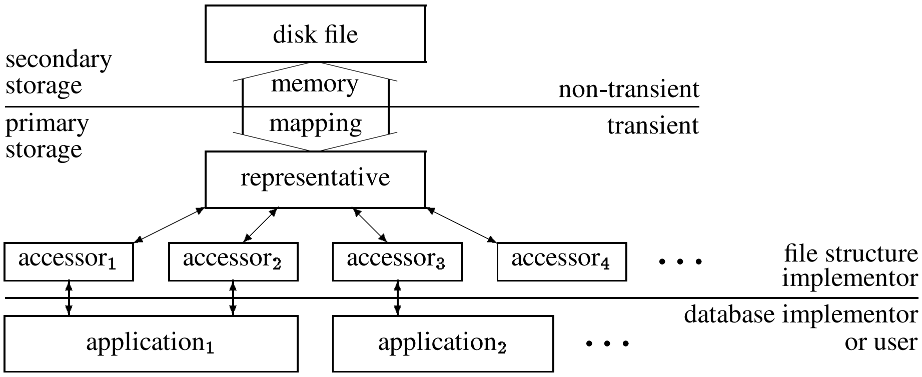

Figure 1: Basic Structure of the Design Methodology

4.1 Representative

A *representative* is responsible for creation and initialization of the file structure for the storage management of access method data in primary storage, for concurrent accesses to the file's contents, and for recovery. Each file structure has a unique representative. In μDatabase, the representative is a UNIX process, which has its own virtual address space in which transient information is maintained and the file is mapped, and its own thread of control. The representative process is created on demand, during creation of a file and for subsequent access by a user, and exists only as long as required by either of these operations.

A representative's memory is divided into two sections: private and shared (see Figure 2). Private memory can only be accessed by the thread of control associated with the UNIX process that created it, i.e. the representative. The disk file is mapped into the private memory while all data associated with concurrent access to the file is contained in the representative's shared memory; such data is always transient. Shared memory is accessible by multiple threads associated with UNIX processes that interact with the representative. There is no implicit concurrency control among threads accessing shared memory; mutual exclusion must be explicitly programmed by the file structure designer using the facilities in μC++.

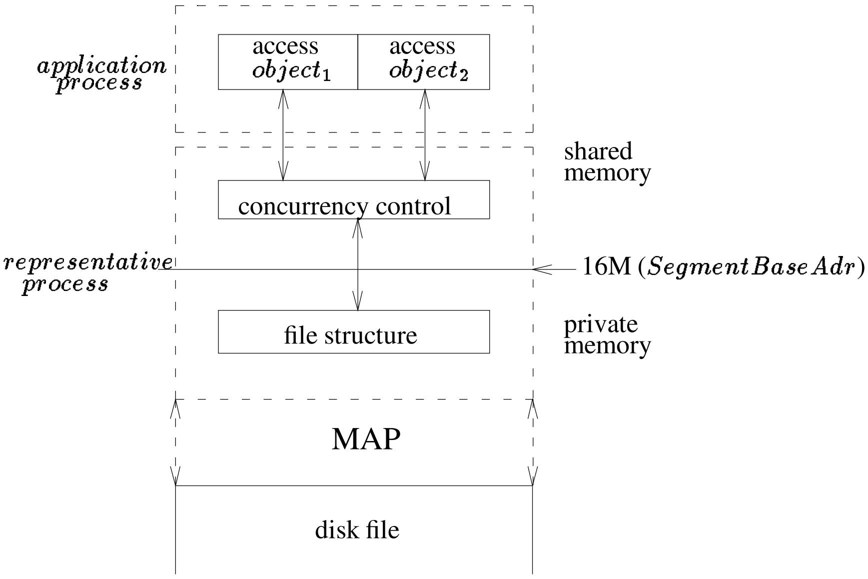

Figure 2: Storage Model for Representative

To allow addresses to be stored directly into the file and subsequently used, the following convention is observed by all representatives: the disk file must be mapped into memory starting at a fixed memory location, called the *Segment Base Address*. The file base address is conceptually the *virtual zero* of a separate segment; this is how μDatabase uses a UNIX process as a separate segment. In μDatabase, the value 16M has been chosen for the Segment Base Address as the starting location of all mapped files; this leaves a sufficiently large space for the application and the representative(s).

An application in μDatabase can have multiple file structures accessible simultaneously. This capability is possible because each representative has its own private mapping area. Figure 3 shows the memory organization of an application using 3 file structures simultaneously. Since each representative has its own segment, relocation of pointers in a file structure is never required. The disadvantage of this approach is that there can never be pointers from the shared area to any of the private mapped areas and vice versa. However, addresses from one file structure can be stored in another file structure, but such addresses can only be dereferenced in the file structure they come from. Hence, either data must be copied out of a file structure to be manipulated by the application and copied back again, or an application light-weight task must migrate to the representatives to perform a series of operations.

4.2 B-Tree Example

To define a file structure, e.g. BTreeFile, an abstract data-type is defined with two operations that are implicitly performed: initialization and termination; no other operations are available. A B-Tree is defined as follows:

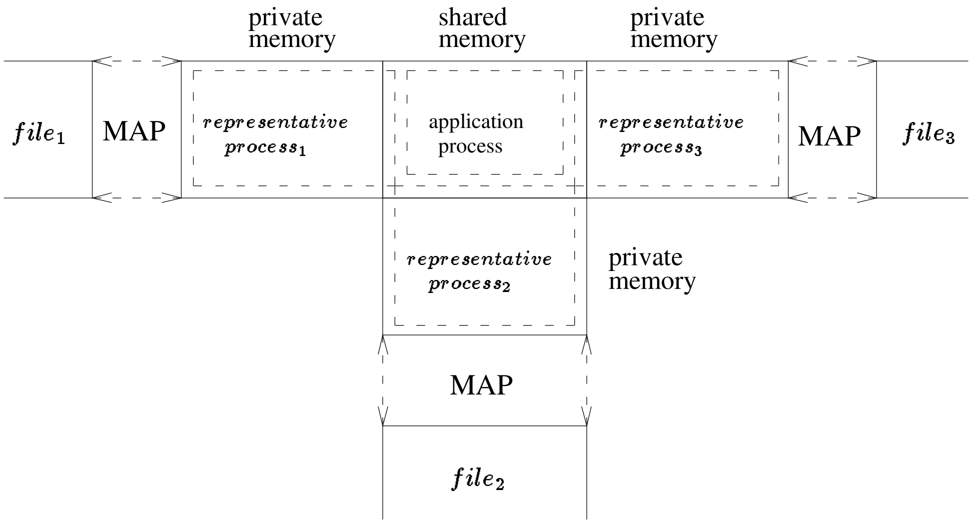

Figure 3: Accessing Multiple File Structures

```
class BTreeFile {
  public:
    BTreeFile( char *DiskFileName, ... ) { initialization code };
    ˜BTreeFile( void ) { termination code };
};
```

The initialization routine BTreeFile and the termination routine ˜BTreeFile are invoked automatically whenever an instance of BTreeFile is created and deleted, respectively. An instance of a B-Tree file structure is created using type BTreeFile, as in: BTreeFile f("StudentData", *other arguments*), where StudentData is the name of the UNIX file in which the data are stored and retrieved from. There are no user visible routines, which ensures that after the declaration of an instance of BTreeFile, the corresponding file structure is not accessible to the user/application program.

4.3 Access

The mechanisms for requesting and providing access to a file structure are provided in the form of another abstract data-type, which is implemented as a class called an *access* class. Declaration of an access class instance, called an *access object*, constitutes the explicit action required to gain access to a file's contents (i.e. create the mapping). Creating an access object corresponds to opening a file in traditional systems but it is tied into the programming-language block structure. As well, the access object contains any transient data associated with a particular access (e.g. the current record pointer), while the representative contains global transient information (e.g. the type of access for each accessor). Because the access object is in the application process, communication between it and the representative process is done by synchronous calls passing data through shared memory. At least one access class must be provided for each file definition. It is possible to have multiple access classes, each providing a distinct form of access (e.g. initial loading, sequential, keyed). It is also possible to have multiple access objects communicating with the same representative. This capability allows an application to have multiple simultaneous views of the data (see Figure 1).

For BTreeFile, the access object is called BTreeFileAcc.

```
class BTreeFileAcc {
  public:
    BTreeFileAcc( BTreeFile *f, char *access ) { initialization code };
    ˜BTreeFileAcc() { termination code };
    read( ... ) { ... };
    ...  { other appropriate access routines };
};
```

To gain read access to a file structure object f, an application program declares an instance of BTreeFileAcc, as follows:

BTreeFileAcc pf(&f, "r"). The pointer to f specifies the file structure that is to be accessed through pf, and "r" specifies the kind of access for concurrency control purposes. Depending on the particular kind of concurrency control, the declaration of the access object may block until it is safe to access the file contents and/or individual access routine calls may block. Once instantiated, the access object can be used by an application to perform operations on the file structure by invoking the public member routines of BTreeFileAcc. For example, in order to read from f, a call is made to the member routine read of BTreeFileAcc, as in pf.read(...). The routine read communicates with the representative to perform the desired operation.

4.4 Generic B-Tree

The polymorphic facilities of a programming language can be applied to generalize the definitions of file structures and to allow reuse of the file structure's implementation by other file structures. A generic B-Tree file structure is presented to demonstrate the basic concept. The template facilities of C++ allow the creation of generic file structures (as in E [RCS89]). The generic B-Tree definition has 2 type parameters and 1 conventional parameter. The type parameters provide the type of the key and the type of the record for the B-Tree. The optional conventional parameter provides the size of the B-Tree nodes in bytes. Each B-Tree instance generated from a generic B-Tree type has 3 conventional parameters: the backing-store UNIX file name, the routine used to compare the keys and the initial space allocated for the B-Tree in bytes. The following creates two specialized B-Trees:

```
BTree<int, Record, 4 Kb> db1( "db1BTree", less ),   at10.0pt// default initial size
     db2( "db2BTree", greater, 30 Kb );              // 30 K initial size
```

Both B-Trees have int keys, Record records and a 4K node size. One instance is sorted in ascending order (less) and the other one in descending order (greater). Unfortunately, this B-Tree instantiation requires the UNIX file name and the name of the comparison routine be re-specified at each subsequent usage of the file structure, which is type unsafe. However, once these two aspects of a file structure are specified correctly, all subsequent access to the database file structure can be statically type-checked.

There are several requirements on the key type, the record type and the comparison routine. As well, some additional routines must be supplied. For example, the type of the key and the record must provide an assignment operator, among other things, and the comparison routine must have a specific type. A complete example showing the creation of a B-Tree and insertion and retrieval of records is presented in Figure 4.

In μDatabase, each file structure can provide range queries using a generator or iterator [RCS89, LAB+81], e.g. BTreeGen. The *generator* is an object whose arguments define the kind of range query and it returns one record at a time from the set of records that satisfy the requirements denoted by the information provided to the generator. The operator ¿¿ returns a pointer to some record within the specified range, but successive records are not normally ordered. If all records in the range have been returned, the NULL pointer is returned. By iteratively invoking the operator ¿¿, the individual records of the range query are obtained.

4.5 Storage Management

In μDatabase, memory is divided into three major levels for storage management: an address space, which is a set of addresses from 0 to N used to refer to bytes or words of memory; a segment, which is a contiguous portion of an address space; and a heap, which is a contiguous portion of a segment. All segments are nested in an address space and all heaps are nested in a segment. Further, since a heap is simply a block of storage, it is possible for heaps to be nested within one another. The form of the address for each level may depend on the storage management scheme at that level.

While there are a large number of storage management schemes possible at each level of nesting, the following three basic schemes are provided in μDatabase: uniform management, the allocation size is the same for the duration of the heap; variable management, the allocation size can vary but each allocation remains that size for its duration (like C's malloc and free routines); dynamic management, the allocation size can vary in size and each allocated area can expand and contract in size after its allocation.

A heap may be accessed in two ways: by the file structure implementor and by a nested heap. For example, the storage management for a B-Tree has 3 levels: the segment, within which uniform-size B-Tree nodes are allocated, within which uniform or variable sized records are allocated. Depending on the particular implementation of the storage manager at each level, different capabilities will be provided. A file structure implementor makes calls to the lowest level (variable storage manager) to allocate records. An expansion object can be passed to the uniform storage manager

```
class Record {                                      // data record
  public:
    float field1, field2;
    Record &operator=( const Record &rhs ) { // define assignment
        field1 = rhs.field1;
        field2 = rhs.field2;
        return( *this ); }
};
int greater( const int &op1, const int &op2 ) { // key ordering routine
    return op1 > op2;
}
void uMain::main() {                                // uMain uC++ artifact
    BTree<int, Record, 4 Kb> db( "testdb", greater, 30 Kb );
    BTreeAccess<int, Record, 4 Kb> dbacc( db ); // open B−Tree
    int key;
    Record rec, *recp;
    // insert records
    for ( key = 1; key <= 1000; key += 1 ) {
        rec.field1 = key / 10.0;
        rec.field2 = key / 100.0;
        dbacc.insert( key, &rec ); }                // static type−checking
    // retrieve records
    for ( BTreeGen<int, Record, 4 Kb> gen(dbacc); gen >> recp; ) {
            uCout << recp−>field1 << "  " << recp−>field2 << endl; }
}
```

Figure 4: Example Program using a Generic B-Tree

to deal with node splitting and other application-specific requirements. If the segment fills with uniform-size nodes, the representative storage manager is called by the uniform storage manager to extend the segment.

The criterion used to judge the general storage management approach is whether it can provide performance that is close to traditional schemes that amalgamate storage management directly with the data structure. Both an independent and integrated storage management B-Tree were constructed and creation tests were run. The results were virtually identical, with timings varying by $\pm$ 2%.

5 Experimental Proof

A number of compelling arguments have been made in [CFW90] and other publications for the use of single-level stores for implementing databases. In spite of these arguments, it is clear there is still resistance and skepticism in the database community. Furthermore, our contention is that mapped files can be used advantageously for building databases not only in the new single-store environment but also in the traditional environment. Traditional databases can be accessed using memory-mapped access methods without requiring any changes to the file structure. In all cases, the mapped access methods should provide performance comparable to traditional approaches while making it much easier to augment the access methods of the file structure in the future by greatly reducing program complexity.

At the start of our work, there was little published experimental evidence available to support the view that memory-mapped file structures could perform as well as or better than traditional file structures. Therefore, it was necessary to implement a number of different memory-mapped file structures and to compare their performance against equivalent traditional ones.

5.1 Experimental Structure

To demonstrate the benefits of memory mapping, different experiments were constructed. The general form of an experiment was to implement a file structure in the traditional and memory-mapped styles, perform retrievals from a file, which is the most common form of access in a database, and compare the results. While every effort was made to keep the two file structures as similar as possible, some system problems precluded absolutely identical execution environments. In particular, the traditional file structures were stored on disk as character-special UNIX files and an LRU buffer manager was used. The memory-mapped files could not be mapped from a character-special file and had to be accessed from the UNIX file system, which performs all I/O in 8K blocks even though the system page size is 4K. Therefore, to make the comparisons equal, all of the file structures had 8K node sizes and all I/O was done in 8K blocks.

All experiments were run on a Sequent Symmetry with 10 i386 processors, which uses a simple page-replacement algorithm. The page-replacement algorithm is FIFO per page table plus a global LRU cache of replaced pages so there is a second chance to recover a page before it is reallocated. The maximum total size of the resident pages for a program is determined by the user and upon exceeding that size, pages are removed from the resident set on a FIFO basis. Upon removal, a page is put into the global cache where it can be reinstated to the resident set if a fault occurs for the page before the page is reused. This page replacement algorithm was matched against an LRU buffer-manager used by the traditional databases.

The execution environment was strictly controlled so that results between traditional and memory-mapped access methods were comparable. First, all experiments were run stand-alone to preclude external interference, except for those experiments that needed a loaded system. Second, the amount of memory for the experiment's address space and the global cache were tightly controlled so that both kinds of file structures had exactly the same amount of buffer space or virtual memory, respectively. The test files varied in size from 6-32 megabytes. The amount of primary storage available for buffer management or paging was restricted so that the ratio of primary to secondary storage was approximately 1:10 and 1:20. These ratios are believed to be common in the current generation of computers, supporting medium (0.1G-.5G) to large databases (1G-4G) but not very large databases.

The following experiments were implemented:

Prefix B^+Tree In this experiment, 100,000 uniformly distributed records were generated whose keys were taken from the unit interval. A record had a variable length with an average of 27 bytes. These records were inserted into a prefix B^+Tree [BU77]. For this B-Tree, 4 query files were generated, where the queries followed a uniform distribution. Each file is described by the tuple (n,m) where n is the number of queries and m is the number of records sequentially read from the B-Tree (range query). For example, (10,1000) means executing 10 queries with each query reading a set of 1000 records sequentially. A 5th query file contained 10,000 exact match queries that follow a normal distribution with mean 0.5 and variance 0.1.

R-Tree The R-Tree [Gut84] is an access method for multidimensional rectangles. It supports *point* queries and different types of *window* queries. A point query asks for all rectangles that cover a given query point whereas a window query asks for all rectangles which enclose, intersect or are contained in a given query rectangle. The window queries are similar to a range query in an ordinary B-Tree. However, there is one basic difference: index pages (internal nodes) are accessed more frequently in the case of the R-Tree than in the B-Tree case.

For this experiment, 2-dimensional data (100,000 rectangles) and queries from a standardized test bed [BKSS90] were taken. The maximum number of data rectangles was limited to 450 in the data pages, and to 455 in the directory pages. The query file consisted of 1000 point queries and 400 each of the three different types of window queries.

Graph To simulate the access patterns found in other data intensive applications (e.g. hypertext or object-oriented databases), a large directed graph was constructed consisting of 64,000 nodes, each of which was 512 bytes. The nodes were grouped into clusters of 64 where nodes in a cluster were physically localized. An edge out from a node had a high probability (85%, 90%, 95%) of referencing a node within the same cluster. Edges leaving a cluster went to a uniformly random selected node. Each experiment consisted of 40 concurrent random walks within the graph, consisting of 500 edge traversals each.

The results of the experiments are presented in Table 1. For each query file, three performance measures were gathered: the CPU time, the elapsed time, and number of pages or buffers read. The CPU time is the total time spend by all processors in a given test run, and hence, the CPU time may be greater than the elapsed time. Multiple processors were used in both traditional and memory mapped experiments. The retrieval application ran on one processor while the access method for the particular file structure ran on another processor. The elapsed time is the real clock time from the beginning to the end of the test run. Both times include any system overhead.

Primary Memory Size 10% of Database Size

Access Method	Query Distr.	Memory Mapped CPU* Time (secs)	Memory Mapped Elapse Time (secs)	Memory Mapped Page Reads	Traditional CPU Time (secs)	Traditional Elapse Time (secs)	Traditional Disk Reads
Prefix B-Tree	1x10,000	35.7	19.7	61	32.2	32.9	53
	10x1,000	35.7	19.5	56	32.5	32.6	58
	100x100	37.5	22.4	147	35.4	35.7	150
	10,000x1	98.1	217.6	8789	240.5	223.6	8746
	normal	91.8	181.0	6777	202.3	183.6	6638
R-Tree	non-point	154.0	174.5	1414	330.4	334.1	1462
	point	109.4	124.1	934	230.5	234.4	896
Network Graph	85% local	318.1	476.1	15294	526.7	458.8	15004
	90% local	271.6	375.5	11278	449.0	370.7	11368
	95% local	207.0	243.8	6584	337.7	254.7	6539

Primary Memory Size 5% of Database Size

Access Method	Query Distr.	Memory Mapped CPU* Time (secs)	Memory Mapped Elapse Time (secs)	Memory Mapped Page Reads	Traditional CPU Time (secs)	Traditional Elapse Time (secs)	Traditional Disk Reads
Prefix B-Tree	1x10,000	35.5	19.5	61	35.3	35.5	117
	10x1,000	35.2	19.6	66	34.2	33.5	131
	100x100	37.0	22.1	155	37.4	36.6	216
	10,000x1	127.8	255.6	9415	260.7	224.1	9723
	normal	126.6	235.8	8250	253.8	217.6	9313
R-Tree	non-point	181.3	227.8	2913	367.1	374.5	3396
	point	136.8	184.5	2647	279.5	289.6	3491
Network Graph	85% local	383.3	565.8	17772	563.4	495.8	16550
	90% local	330.3	462.1	13602	484.0	403.9	12781
	95% local	264.9	316.6	8338	361.9	276.1	7400

CPU times may be greater than elapse time because multiple CPUs are used.

Table 1: Access Method Comparison : Node Size 8K

The results of the experiments confirm the conjecture that performance of memory-mapped file structures is equivalent or better than traditional file structures. For the read operations, the memory-mapped access methods are comparable ($\pm$ 10%) to their traditional counterparts. An exception occurs when the LRU buffer space is only 5% of the file size for sequential reads because the LRU algorithm is suboptimal in this case while the FIFO page-replacement algorithm is near optimal. For the CPU times, the memory-mapped access methods are generally better than the traditional ones because there is less time spent doing buffer management. For the elapsed times, the memory-mapped access methods are comparable ($\pm$ 10%) to their traditional counterparts. An exception occurs when memory-mapped access methods perform small sequential reads because the FIFO page-replacement algorithm is near optimal in this case. All of the results show that the Sequent page replacement scheme performed comparably to the LRU buffer-manager.

To verify the conjecture on the expected behavior of mapped access methods on a loaded machine, the previous B-Tree experiments were run during a peak-load period of 20-30 time-sharing users on the Sequent. The memory mapped and traditional B-Tree retrievals were started at the same time (3:00pm) and so were competing with each other as well as all other users on the system. The two file structures were on different disks accessed through different controllers so the OS could not share pages and retrievals were not interacting at the hardware I/O level. However, the amount of global cache could not be restricted during the day, so if there was free memory available, the memory-mapped access method would use it indirectly. Table 2 shows the averages of 5 trials. As can be seen, there was a difference only when there were a significant number of reads. In those cases, the memory-mapped access methods make use of any extra free memory to buffer data. This is particularly noticeable for the normal distribution because any extra memory significantly reduced the pages read, and hence, the elapse time. Clearly, the LRU buffer manager could be extended to dynamically increase and decrease buffer space depending on system load, but that further complicates the buffer manager and duplicates code in the operating system.

Primary Memory Size 10% of Database Size

		Memory Mapped			Traditional		
Access Method	Query Distr.	CPU* Time (secs)	Elapse Time (secs)	Page Reads	CPU Time (secs)	Elapse Time (secs)	Disk Reads
Prefix B-Tree	1x10,000	35.8	21.6	60	34.0	35.7	53
	10x1,000	36.1	21.8	56	34.8	36.8	58
	100x100	37.4	25.24	143	37.0	38.68	150
	10,000x1	111.2	277.0	6677	263.4	263.3	8746
	normal	97.82	134.5	2063	221.5	217.0	6638

Table 2: Peak Load Retrievals : Node Size 8K

6 Parallelism

Currently, μDatabase allows a file structure designer to build whatever form of concurrency control is appropriate. Concurrency control can be specified at a low-level, where semaphores are used to protect data, or at a high-level, where light-weight server tasks control access to data. While concurrency control is often tied into a particular data structure, we believe it is possible to provide some general concurrency abstractions to the file structure designer to aid in this process. A number of different concurrency techniques are being studied that provide two different forms of parallelism. *Backend concurrency* deals with the I/O bottleneck, a file structure is partitioned across multiple disks and access is performed in parallel. *Frontend concurrency* allow a number of requests to execute in parallel if the requests access data in different areas of the database. The question to be addressed is how to use memory mapping with both backend and frontend concurrency.

6.1 Backend Concurrency

Backend concurrency attempts to deal with the CPU-I/O bottleneck by partitioning data across multiple disks and then accessing the data in parallel [PGK88]. Exact match queries usually cannot take advantage of parallelism possible from partitioning because there is usually only one disk access to service the request. Range queries can take advantage of the parallelism possible from partitioning if the data is distributed so that portions of the range can be accessed in parallel. A range query may be broken down into a number of smaller range queries so that each can be executed in parallel. Similarly, if the file structure is aware of the access pattern of different blocks, it can employ pre-reading techniques to increase the parallelism in reading blocks of data from the disk. In general, the records returned from a range query are unordered. If records must be returned in a specific order, that can significantly reduce the amount of parallelism. In μDatabase, the generator types for each file structure can manage all concurrent retrieval of records implicitly (see Figure 4)

In the following discussion, the general concern is not about access to the index portion of the file structure. Normally the index is relatively small so that most of it remains resident in main memory, and consequently, does not play a significant role as far as disk accesses are concerned.

6.1.1 Generic Backend Concurrency Algorithm

Once a file structure is partitioned, a retrieval algorithm can take advantage of the potential parallelism, but only if sufficient hardware is available. First, the disks must be able to be accessed in parallel, which implies that there must be multiple disk controllers. Second, if multiple processors are available, they must be able to be used to perform any file-structure administration in parallel with the application processing the records from the range query. Both of these hardware requirements were satisfied by our Sequent computer.

The algorithm used for backend concurrency is as follows. For a file structure partitioned across N disks, the N disk files are memory mapped into one contiguous segment. Then M (a control variable) kernel threads (UNIX processes) are created that all share the data segment containing the mapped file. $N+1$ light-weight tasks are created to perform the retrieval requests and they execute on the M kernel threads. N of the tasks are retrievers and the $(N+1)^{th}$ task is the *leaf retrieval administrator* (LRA). For each generator created, a buffer is allocated by the generator, which is shared between the application and the file structure. As well, another task, the *file structure traverser*, is generated, which partitions the range query. The size of the buffer can be specified as an optional parameter when creating the generator. The default buffer size is 32K bytes. The traverser task assumes the responsibility of organizing the buffer space in the form of a sharable buffer pool in some suitable manner. Then the traverser task searches the index structure finding the leaf nodes that contain records in the range. For each leaf node, the traverser communicates with the LRA specifying the leaf, number of records in the leaf, and the buffer pool. The LRA farms out the generator requests to its retrieval tasks. A retrieval task accesses the specified leaf page, allocates a buffer from the buffer pool, and copies as many records as will fit from the leaf page to the buffer. The last step is repeated until all the records have been copied into buffers and then the retriever task gets more work from the LRA. The structure of this algorithm is illustrated in Figure 5. This structure ensures that the only bottleneck in the retrieval is the speed that the buffer can be filled or emptied. In general, an application program can keep ahead of a small number of disks (1-7 disks). This generic backend concurrency algorithm can be used for different file structures by specializing the file structure traverser and the component responsible for processing of individual leaves to extract information.

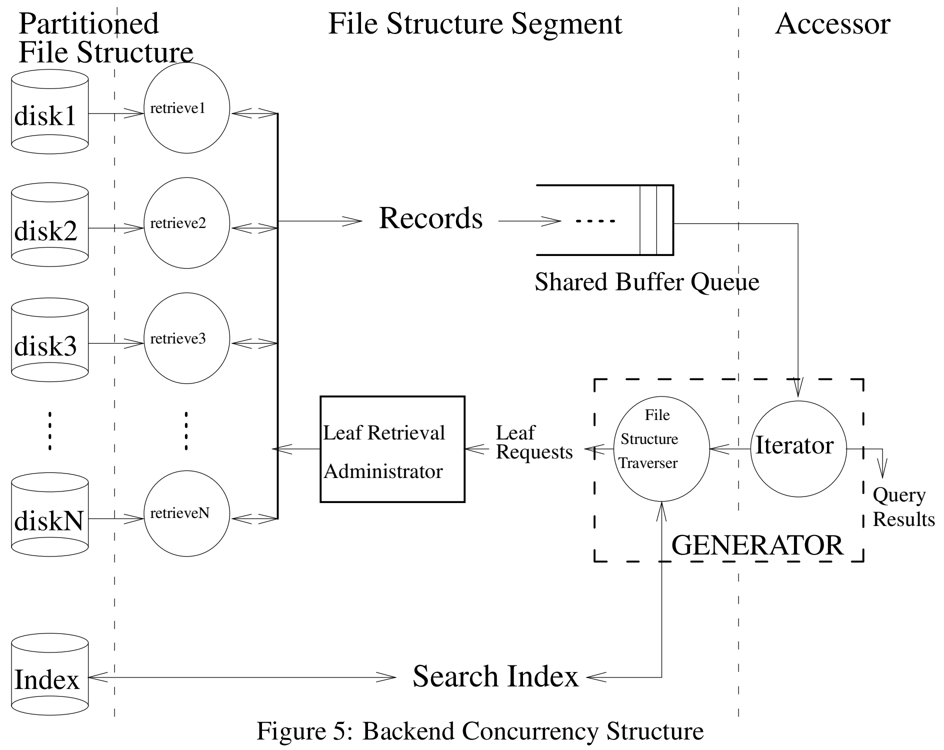

Figure 5: Backend Concurrency Structure

6.1.2 Experimental Analysis of Partitioned B-Tree

The machine used for these experiments was the same Sequent Symmetry with 8 disk drives, of which 4 were used. There were 2 disk controllers, each with 2 channels. The drives were equally divided between the controllers. The experiment was 1000 range queries with each query consisted of reading a random number of sequential records starting at a randomly selected initial key. The average query size was 2000 records. Two partitioned B-Trees were tested, one created using a round-robin partitioning (each block is created on the next disk) and one created using the Larson-Seeger algorithm [SL91]. The partitioned experiments were performed with 1–4 partitions and the application program received each record but did no processing on the record. The results of the experiments appear in the graphs of Figure 6. The largest decrease in elapsed time is from 1 to 2 partitions because there are 2 controllers. After that, the elapse time increases because of contention on the two controllers.

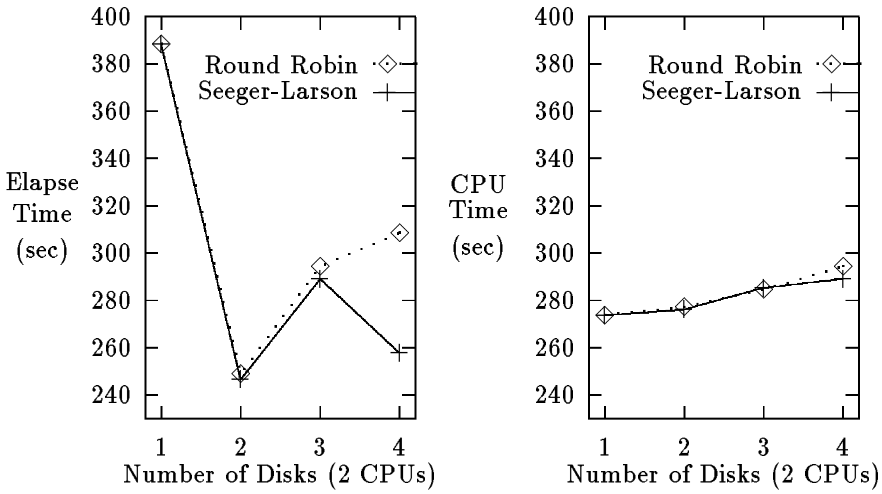

Figure 6: Backend Concurrency with B-Trees

6.2 Frontend concurrency

Here the concern is with allowing multiple client accessors to simultaneously traverse and manipulate the file structure. Currently, μC++ provides a number of language mechanisms for a file designer to build concurrency control. Many options will be built, tested and provided as part of μDatabase tool kit, however these will be used to build file-structure specific concurrency control. It is also our intention to study and develop a general purpose low-level concurrency control facility that will be automatically available to applications written in μDatabase. For example, allowing multiple versions of data to co-exist allows a high degree of concurrent and can be implemented in a general way.

7 Recovery Control

Implementing recovery is difficult in memory mapping and a satisfactory solution is still a research issue. If there is operating-system support to pin pages, traditional schemes can be used (however, with all the associated disadvantages). With no operating-system support, new techniques must be developed. We will be examining the use of dual memory maps to allow shadow write pages. One mapping represents the consistent database, which can be read at any time. The shadow mapping is for pages that are currently being modified. By precisely controlling when the shadow pages are copied back to the consistent mapping, it is possible to mimic traditional recovery schemes without operating-system support. The main problem to overcome is premature writing of modified pages by the operating system.

8 Related Work

The earliest use of memory mapping techniques (or a single-level store) can be found in the Multics system [BCD72]. In recent times a number of efforts have been made to use memory mapping. The systems described below are most closely related to μDatabase.

Objectstore Database System Objectstore shares a number of goals and objectives with μDatabase. However, Objectstore differs significantly from μDatabase in how the goals and objectives are achieved. In Objectstore, only the currently accessed pages used by a given transaction are mapped into the address space of the application. This approach introduces a limit on the number of different data pages that can be used simultaneously by any single transaction; large operations may have to be broken into a series of smaller transactions. In μDatabase, an entire file structure is mapped into an individual segment. This approach limits the size of any single file structure to be less than the virtual space supported by the available hardware; large file structures have to be split into smaller ones. There is, however, no restriction on how much data a single transaction can access simultaneously.

The approach used in Objectstore results in an inferior solution to the problem of accessing multiple file structures. Objectstore maps pages of all the databases used in an application into the same address space. Each page to be used is dynamically allocated a virtual address where it is mapped; pointers have to be dynamically relocated, which requires some portion of the type system to be available at runtime. Also, the need to relocate pointers has the potential of degrading performance of the database.

Cricket: A Mapped, Persistent Object Store Cricket uses the memory management primitives of the Mach operating system to provide the abstraction of a "shared, transactional single-level store that can be directly accessed by user applications" [SZ90a, p. 89]. Cricket follows a client/server paradigm and, upon an explicit request, maps the database directly into the virtual space of the client application. The fundamental difference from μDatabase is that the mapping takes place in the address space of the application, and hence, only one database at a time can be used by an application. Indeed, the concept of a disk file to group related objects in one collection is not a basic entity in Cricket and it takes the view that everything that an application needs to use is placed in a single large persistent store. We feel that this will lead to a certain amount of awkwardness in organizing various components of data and in sharing pieces of data across different projects. More importantly, this approach will not be able to handle partitioning of data across multiple disks adequately.

Paul Wilson's work In [Wil91], Paul Wilson describes a scheme that uses pointer swizzling at page fault time to support huge address spaces. The basic scheme is very similar to the one employed by Objectstore except that in Wilson's scheme pointers on secondary store can have a format different from the pointers in primary storage. Wilson's scheme requires a special page fault handler for translating (swizzling) persistent pointers into transient pointers at execution time, which requires runtime type information. Since some of the pointers in a page can refer to pages that have not yet been made available, the translation of these pointers requires that all the referent pages be *faulted* as well. To prevent a cascade of I/O operations, Wilson's scheme only reserves the addresses for these extra pages in the page table instead of actually mapping them to primary storage. However, this solution underutilizes the address space and an application can potentially run out of addresses. Wilson suggests periodically invalidating all the mappings and rebuilding them to deal with this problem. Furthermore, objects that cross page boundaries require additional language support. Wilson's scheme is a clear winner for applications that require extremely large persistent address spaces using existing virtual memory hardware. However, the scheme is complex and may result in significant overhead, especially for applications with poor locality of references. Finally, Wilson's approach has the same problems as Objectstore with regard to dynamic relocation and multiple accessible databases.

The Bubba database system The designers of Bubba [BAC$^+$90, CFW90], a highly parallel database system developed at MCC, exploited the concept of a single-level store to represent objects uniformly in a large virtual address space. The focus of Bubba was on developing a scalable *shared-nothing* architecture which could scale up to thousands of hardware nodes and the implementation of a single-level store was only a small, though important, portion of the overall project. The current design of μDatabase is based on a multiprocessor shared-memory architecture and is not intended to be used in a distributed environment. In Bubba, the Flex/32 version of AT&T UNIX System V Release 2.2 was extensively modified to build a single-level store, which makes their store highly unportable.

9 Conclusion

We have shown that memory mapping is an attractive alternative for implementing file structures for databases. Memory-mapped file structures are simpler to code, debug and maintain, while giving comparable performance when used stand-alone or on a loaded system than for traditional databases. Further, buffer management supplied through the page-replacement scheme of the operating system seems to provide excellent performance for many different access patterns. Our design for structuring the low-level portions of a DBMS for memory mapping provides the necessary environment to implement concurrency control and recovery. Finally, these benefits can be made available in tool kit form on any UNIX system that supports the `mmap` system call. Currently, μDatabase is only missing recovery facilities and these will be added in the near future.

References

[BAC+90] H. Boral, W. Alexender, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez. Prototyping Bubba, A Highly Parallel Database System. *IEEE Trans. on Knowledge and Data Eng.*, 2(1):4–24, March 1990.

[BCD72] A. Bensoussan, C. T. Clingen, and R. C. Daley. The Multics Virtual Memory: Concepts and Design. *Communications of the ACM*, 15(5):308–318, May 1972.

[BDS+92] P. A. Buhr, Glen Ditchfield, R. A. Stroobosscher, B. M. Younger, and C. R. Zarnke. μC++: Concurrency in the Object-Oriented Language C++. *Software—Practice and Experience*, 22(2):137–172, February 1992.

[BKSS90] N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 322–331, 1990.

[BU77] Rudolf Bayer and Karl Unterauer. Prefix B-Trees. *ACM Transactions on Database Systems*, 2(1):11–26, March 1977.

[BZ86] P. A. Buhr and C. R. Zarnke. A Design for Integration of Files into a Strongly Typed Programming Language. In *Proceedings IEEE Computer Society 1986 International Conference on Computer Languages*, pages 190–200, Miami, Florida, U.S.A, October 1986.

[BZ89] P. A. Buhr and C. R. Zarnke. Addressing in a Persistent Environment. In John Rosenburg and David Koch, editors, *Persistent Object Systems*, pages 200–217, Newcastle, New South Wales, Australia, January 1989. Springer-Verlag. Workshops in Computing, Ed. by Professor C. J. van Rijsbergen, QA76.64.I57.

[CAC+84] W. P. Cockshott, M. P. Atkinson, K. J. Chisholm, P. J. Bailey, and R. Morrison. Persistent Object Management System. *Software—Practice and Experience*, 14(1):49–71, 1984.

[CFW90] George Copeland, Michael Franklin, and Gerhard Weikum. Uniform Object Management. In *Advances in Database Technology - EDBT'90*, volume 416, pages 253–268, Venice, Italy, March 1990. Springer-Verlag.

[Gut84] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 47–57, 1984.

[IBM78] *System/38 Services Overview*. IBM, 1978.

[LAB+81] Barbara Liskov, Russell Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. *CLU Reference Manual*, volume 114 of *Lecture Notes in Computer Science*. Springer-Verlag, 1981.

[LLOW91] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The Objectstore Database System. *Communications of the ACM*, 34(10):50–63, October 1991.

[MBC+89] R. Morrison, A. Brown, R. Carrick, R. Connor, A. Dearle, and M. P. Atkinson. The Napier Type System. In John Rosenberg and David Koch, editors, *Persistent Object Systems*, pages 3–18, University of Newcastle, New South Wales, Australia, January 1989. Springer-Verlag. Workshops in Computing, Ed. by Professor C. J. van Rijsbergen, QA76.64.I57.

[Mip91] *MIPS R4000 Microprocessor User's Manual*. MIPS Computer Systems Inc, 1991.

[Mos90] J. Moss. Working with Persistent Objects: To Swizzle or Not to Swizzle. Technical Report CS Technical Report 90-38, Computer Science Department, University of Massachusetts, May 1990.

[Org72] E. I. Organick. *The Multics System*. The MIT Press, Cambridge, Massachusetts, 1972.

[PGK88] D. A. Patterson, G. Gibson, and R. H. Katz. A Case for Redundant Arrays of Inexpensive Disks(RAID). In *Proceedings of the 1988 ACM SIGMOD*. ACM, June 1988.

[PS-87] The PS-Algol Reference Manual, 4th Ed. Technical Report PPRR 12, University of Glasgow and St. Andrews, Scotland, June 1987.

[RCS89] Joel E. Richardson, Michael J. Carey, and Daniel T. Schuh. The Design of the E Programming Language. Technical Report CS-TR-824, Computer Science Department, University of Wisconsin-Madison, Madison, Wisconsin, 53706, February 1989.

[RKA92] J. Rosenberg, J. L. Keedy, and D. A. Abramson. Addressing Mechanisms for Large Virtual Memories. *The Computer Journal*, 35(4):369–375, August 1992.

[SL91] Bernhard Seeger and Per-Ake Larson. Multi-Disk B-trees. In *Proceedings of the 1991 ACM SIGMOD*, pages 436–445, Denver, Colorado, USA, June 1991. ACM.

[STP+87] Alfred Z. Spector, D. Thompson, R. F. Pausch, J. L. Eppinger, D. Duchamp, R. Draves, D. S. Daniels, and J. L. Bloch. Camelot: A Distributed Transaction Facility for Mach and the Internet - An Interim Report. Technical Report CMU-CS-87-129, Carnegie Mellon University, 1987.

[Sun90] *System Services Overview*. Sun Microsystems, 1990.

[SZ90a] Eugene Shekita and Michael Zwilling. Cricket: A Mapped, Persistent Object Store. In A. Dearle et al, editor, *Implementing Persistent Object Bases: Principles and Practise*, pages 89–102. Morgan Kaufmann, 1990.

[SZ90b] M. Stumm and S. Zhou. Algorithms Implementing Distributed Shared Memory. *IEEE Computer*, 23(5):54–64, May 1990.

[TRY+87] A. Tevanian, Jr., R. F. Rashid, M. W. Young, D. B. Golub, M. R. Thompson, W. Bolosky, and R. Sanzi. A Unix Interface for Shared Memory and Memory Mapped Files Under Mach. In *Proceedings of the Summer 1987 USENIX Conference*, pages 53–67, Phoenix, Arizona, June 1987. USENIX Association.

[vO90] Peter van Oosterom. *Reactive Data Structures for Geographic Information Systems*. Ph.D. Thesis, Dept. of CS, Leiden University, December 1990.

[WF90] K.L. Wu and W.K. Fuchs. Recoverable Distributed Shared Virtual Memory. *IEEE Transactions on Computers*, 39(4):460–469, April 1990.

[Wil91] Paul R. Wilson. Pointer Swizzling at Page Fault Time: Efficiently Supporting Huge Adrress Spaces on Standard Hardware. *Computer Architecture News*, 19(4):6–13, June 1991.