Virtual Files:

a Framework for Experimental Design

by

George D. M. Ross

Ph. D.

University of Edinburgh

1983

for Rhoda

Table of Contents

List of Figures

List of Tables

Abstract

The increasing power and decreasing cost of computers has resulted in them being applied in an ever widening area. In the world of Computer Aided Design it is now practicable to involve the machine in the earlier stages where a design is still speculative, as well as in the later stages where the computer's calculating ability becomes paramount. Research on database systems has not followed this trend, concentrating instead on commercial applications, with the result that there are very few systems targeted at the early stages of the design process. In this thesis we consider the design and implementation of the file manager for such a system, first of all from the point of view of a single designer working on an entire design, and then from the point of view of a team of designers, each working on a separate aspect of a design.

We consider the functionality required of the type of system we are proposing, defining the terminology of *experiments* to describe it. Having ascertained our requirements we survey current database technology in order to determine to what extent it meets our requirements. We consider traditional concurrency control methods and conclude that they are incompatible with our requirements. We consider current data models and conclude that, with the exception of the persistent programming model, they are not appropriate in the context required, while the implementation of the persistent programming model provides transactions on data structures but not experiments.

The implementation of experiments is considered. We examine a number of potential methods, deciding on *differential files* as the one most likely both to meet our requirements and to have the lowest overheads. Measurements conducted on both a preliminary and a full-scale implementation confirm that this is the case. There are, nevertheless, further gains in convenience and performance to be obtained by exploiting the capabilities of the hardware to the full; we discuss these in relation to virtual memory systems, with particular reference to the *VAX/VMS* environment.

Turning to the case where several designers are each working on a (nearly) distinct part of a design, we consider how to detect conflicts between experiments. Basing our approach on *optimistic concurrency control* methods, we show how read and write sets may be used to determine those areas of the database where conflicts might arise. As an aside, we show how the methods we propose can be used in an alternative approach to optimistic concurrency control, giving a reduction in system overheads for certain applications. We consider implementation techniques, concluding that a differential files approach has significant advantages in maintaining write sets, while a two-level bitmap may be used to maintain read sets efficiently.

Acknowledgements

A large number of useful discussions with the members of staff and students of the Department of Computer Science, both past and present, contributed to the research described here, particularly with Malcolm Atkinson and the members of the Data Curator group.

The *Rbase* system (chapter 4 and appendix A) was provided by Austin Tate. Gordon Brebner supplied the random number generator used in evaluating the block-based virtual file implementation (chapters 5 and 6). Paul MacLellan conceived the exception handling mechanism used in the *Shines* system (chapter 7 and appendix C) and performed a substantial part of the implementation.

This work was supported in part by the Science and Engineering Research Council, award number 79502150, in conjunction with British Aerospace under the *CASE* scheme.

Chapter One
The Problem

1.1 Computers and design

Computer Aided Design (CAD) has had a great impact on the design and implementation of large scale systems, partly through the computer's ability to perform tasks which, while they could be performed by humans, are essentially repetitive applications of a set of fairly simple rules which can be followed by a machine with much less chance of error, partly through its ability to perform rapid computation, which allows tasks to be performed which would be outwith the capacity of humans to perform in a reasonably short time scale, and partly through its ability to store and organise large bodies of information in a way which allows integrated design to take place on a much larger scale than would otherwise be possible.

- The *ESDL* system [Smith 80a, Smith 80b] for aiding the design of logic circuits using "discrete" components such as the 7400 series logic family, microprocessors, etc., makes use of the computer's computational abilities. The circuit being designed is coded up in a textual form and compiled. At this stage logical errors may be detected. These might include semantic faults, such as having too high a fanout from a gate, or syntactic errors, such as attempting to connect to non-existent signals or pins. Once the circuit is logically correct it can be simulated. This will indicate where problems are likely to occur, and will indicate the performance characteristics to be expected from the hardware once it is constructed, a task which would be beyond the abilities of humans. At this stage the designer may decide to modify the design if it does not perform to his expectations. Finally, if he is satisfied with his design, a wire-wrap schedule may be produced and the hardware constructed.

- In the *ILAP/CIF* system for VLSI design [Hughes 82] the design is coded up, in a high-level language such as *Imp* or *Pascal*, into a series of calls on procedures in the *ILAP* library. The use of a high-level language facilitates a hierarchical design process, as more complex objects can be constructed from simpler ones by making use of procedures with parameters to define the simpler objects, and constructs such as loops and conditions to manipulate their use in the larger objects. Commonly used objects, such as pads and PLAs, constructed in this way, would appear in the system library. The program is compiled, and, when run,

> generates *CIF*, a textual representation of the chip geometry. This *CIF* is interpreted, and may be further processed or used to drive a graphics device such as a colour raster-scan display, a plotter or, when the design is complete, a mask-making machine. Simulation is still possible, but requires substantially more computing resources than in the previous example since it is being performed at a lower level than for *ESDL*. Design rule checking can also be performed: this will show if the designer has violated any of the rules on, for example, mimimal spacings and overlaps, which are intended to decouple the design process from the actual fabrication technology of the final chip. The problems inherent in this approach, however, has led to the development of special-purpose languages such as *Scale* [Buchanan 82, Marshall 83].

In both these examples the systems take as input a textual representation of the design and produce as their final output a sequence of instructions for manufacturing the end product, with some optional intermediate processing intended to indicate whether the design is likely to function as intended. These systems do not allow the design to be manipulated once it has been input, although the designer clearly has the option of altering the design and modifying the textual representation in the light of the outputs from these systems. They are aimed at the later stages once a design is reasonably complete, rather than at the earlier stages when it is still being produced, a much more volatile phase. Consider the advantages to the designer if he could use the abilities of the computer at an earlier stage in the design process, where he was trying out various possible solutions on the back of his proverbial envelope: the ability to apply some of the tools which would normally be reserved for use with a more stable design could prevent him from being led up a blind alley; he could develop several possible alternative designs and compare them more easily to see which was best; and the computer's ability to organise the storage of his designs for him would mean that he could be much more sure that his current attempts would integrate properly, both with each other and with the surrounding design context. It is this latter problem of organising the storage of designs that we will concentrate on.

We will tackle the problem in two stages: firstly we will consider the case of a single designer working on a single aspect of a design; only when we are satisfied that this can successfully be handled will we turn to the case of several designers all working concurrently on the same design project.

1.2 Characterising the design process

What, then, characterises the early stages of the design process? The answer is that the designer is experimenting, trying possible approaches to the problem, evaluating them, further developing the promising ones and rejecting the unpromising ones until finally a solution is obtained which best meets the desired specification.

The path towards the final solution need not (and indeed probably will not) have been particularly direct. Typically, several approaches will have been evaluated, unless the problem is sufficiently well understood that a known tried and trusted method can be used. The evaluation of these trial approaches will have involved the definition of subproblems, each of which will have been solved by experimenting with possible solutions and eliminating the unpromising ones, and so on recursively. Further, the resulting "solution tree" need not have been generated in a tidy manner, but rather the designer may have been unsure which approach was best and developed several branches in parallel, taking each a small step forward in turn, until it became clear which was going to turn out best. The designer may even have turned to another aspect of the design after encountering a particularly difficult part, in the hope that inspiration might strike. Only towards the end of the process, when unfruitful branches of the tree have been eliminated, will a reasonably coherent design materialise for further processing.

A system intended for such a design environment must be able to handle such a tree of designs, creating and deleting the branches as the design progresses. It must impose no constraints on the order of access to parts of the tree or on their lives,[1] thereby allowing the designer to work on whichever part of the design he wishes.

At each node of the tree the database should appear to contain all the objects defined at its own node and all parent nodes. The effect of this should be such that the designer appears to be modifying a local copy of the database which would result from following the design back to the root, uncontaminated by other design branches.

1.3 The design tree – a closer look

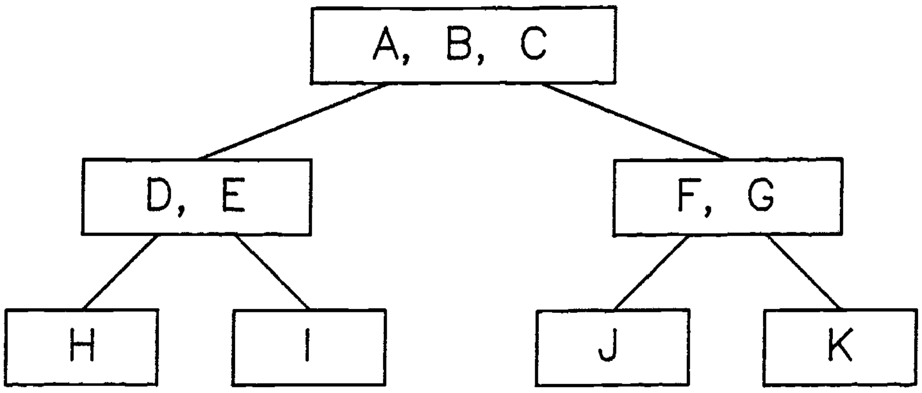
Consider the simple example of a design tree given in figure 1-1. At the base level the design consists of items *A*, *B* and *C*. At the next level the designer has tried two possibilities, one of which adds *D* and *E* to the design, the other adding *F* and *G*. From this level there appear to be two independent databases, one containing *A*, *B*, *C*, *D* and *E*, the other containing *A*, *B*, *C*, *F* and *G*. Similarly, at the final level there appear to be four independent databases, one of which contains *A*, *B*, *C*, *F*, *G* and *J*. The tree is shown as binary in the example, but in practice may be of any shape.

To ensure that the designs remain properly independent of each other it is essential that all modifications to the database take place at the most "leafward" design tree node. The value of an object is then determined by the following rule: if an object appears in a node then the value it has there takes precedence over the values it may have in parent nodes; further, an object may be deleted in a node while remaining in a parent node: in this case the precedence of a leaf over its parent results in the object appearing deleted from that node, but still appearing to exist when viewed from parent nodes.

If, in the example tree above, object *C* had been deleted (indicated by \C) and object *F* modified (indicated by *F'*) at the node containing *J*, then the tree would be modified into figure 1-2. From this node the database would then appear to contain *A*, *B*, *F'*, *G* and *J*

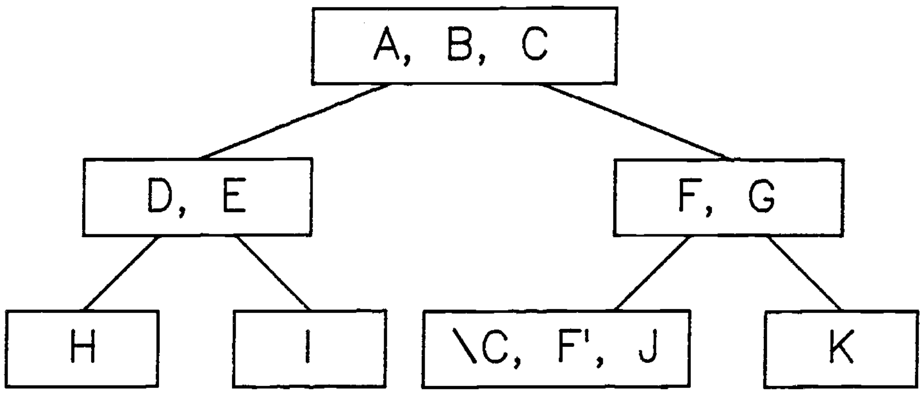
[1]apart from the obvious one concerning nodes whose parents have been deleted or modified so as to render their offspring invalid

Figure 1-1: A simple design tree



(and not *C*). From all other nodes the database appears unchanged, eg from the adjacent node the database contains *A, B, C, F, G* and *K* as before, and similarly for the branches containing *H* and *I*.

Figure 1-2: Modified design tree

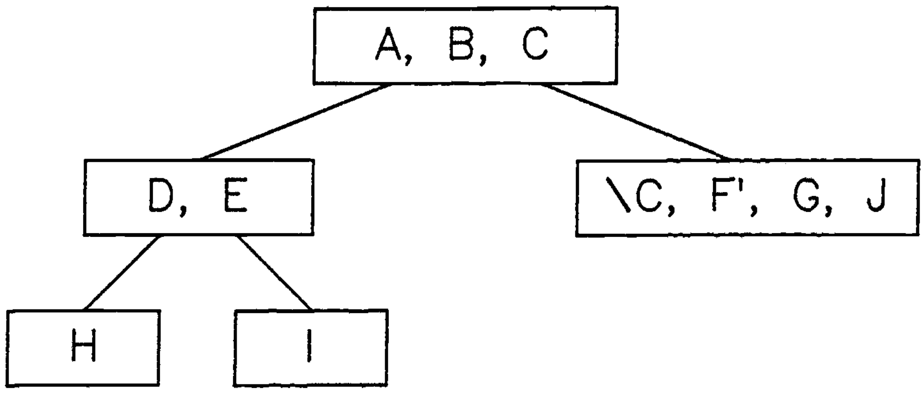


Suppose, now, that the designer decides that this part of the design containing *F'* is superior to the one containing *K*. The node containing *K* is deleted and the nodes containing *G* and *J* are merged to give a new design node, both physically, in the database, and logically. The resulting tree appears as in figure 1-3.

From this new node the database appears to contain *A, B, F', G* and *J*, as it did from the leaf node before merging. The design containing *K* now no longer exists, but the designs containing *H* and *I* have been unaffected by the alterations to the other branch in the last two steps.

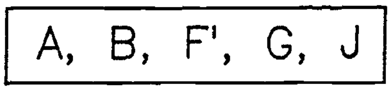
The designer may now decide that the design containing *F'* is superior to the designs

Figure 1-3: Merged design tree



containing *H* and *I*, and that this is to be the final version. The branches containing the unwanted designs are deleted and the branch containing the final design is merged into the base node, giving figure 1-4.

Figure 1-4: The final design tree



1.4 Some terminology

We have seen that each leaf of the design tree exists because the designer has decided to try out a possible solution to (part of) the design. Solutions are experimented with in an attempt to find one which meets the target constraints and specifications. This results in a tree structure for the database holding the design, with leaves and branches being created and deleted as the design progresses.

We define the following terminology: each related set of changes, contained in a new leaf added to the database tree, constitutes an *experiment*; the act of adding a new experiment is called *opening*, that of deleting an unwanted experiment *abandoning*, and that of merging an experiment with its parent *committing*. The designer moves from one node of the design tree to another by *selecting* the one he wishes to work with, thereby *suspending* the previous design and *resuming* the new one. These terms are motivated by analogy with transactions (see page 12). Experiments of experiments are said to be *sub-experiments*. The provision of sub-experiments is both logically necessary from the point of view of modelling the design tree and practically useful since the software in higher layers of the system can be made much more modular, no account needing to be taken, when a new experiment is opened, of whether there is an experiment currently open or not.[1] Note that it is not meaningful to commit or abandon an experiment while there are sub-experiments open on it, or to write to such an experiment except by committing one of its sub-experiments. This terminology is summarised in table 1-1.

Table 1-1: Summary of terminology relating to experiments

Opening	The act of creating a new leaf on the design tree for a related collection of experimental updates. A name is associated with the new experiment in order that it may be referred to during its (potentially long) life.
Committing	A (named) experiment is merged with its parent, so that all modifications which were made in it now take effect in this more general context. This done, the experiment ceases to exist.
Abandoning	A (named) experiment is consigned to oblivion, any updates being lost forever.
Selecting	A (named) experiment is chosen to provide the context in which the designer wishes to operate.
Suspending	When a new experiment is selected the old one is suspended. It will stay in this state until it is resumed, abandoned or committed.
Resuming	The act of selecting and continuing work in a previously suspended experiment.
Sub-experiment	An experiment may have further experiments open on itself. These are known as sub-experiments. There may, of course, be sub-sub-experiments, and so on...

The designer will wish not to be constrained to particular access routes to experiments, but rather will wish to access whichever one he chooses. To facilitate this, and to allow the designer to suspend the design while preserving its state for future resumption, experiments will require to be named. Furthermore, since a design may be worked on for some considerable time before a finished product emerges it is essential that the state of currently-open experiments be preserved in a way that is both non-volatile and does not interfere with any other potential users of the host system.

[1]The corresponding facility, sub-transactions, is not normally provided, although it would be exceedingly useful, both from the point of view of modularity of software, and since it would allow more convenient interactive incremental updating to a database.

1.5 Related work

PIE [Bobrow 80, Goldstein 80a, Goldstein 80b], a personal information environment implemented in *Smalltalk* [Krasner 81, Xerox 81], has many features in common with experiments as outlined above. In *PIE*, values are dependent on the *context* in which they were defined, thus allowing the parallel definition of a number of alternative designs. The idea of contexts is derived from *CONNIVER* [Sussman 72], where they were introduced to try to solve the problem, found in *PLANNER*, that when a particular goal in a method failed all information as to why it failed was lost. Leaving a tree of contexts allows the user to determine why the goal failed, and to use the information gained to direct the subsequent operation of the method.

The assignment of a value to a property is done in a *layer*, with retrieval being done by looking up the value of an attribute layer-by-layer: if a value is asserted from the first layer of a context then that value is returned; otherwise the next layer is examined, the process being repeated until the layers are exhausted. Contexts are extended by creating a new layer. The description of a context is also context-dependent, with the result that a super-context may be created which can affect the values returned by the layers of a number of sub-contexts. The system permits *contracts* between objects, implemented by means of explicit or implicit attached procedures; thus, for example, a constraint that two values must always be equal would be transformed by the system into an attached procedure to enforce the constraint. Contracts are enforced when a layer is closed, the assumption being that the values defined in the layer then form part of a consistent whole.

Layers and contexts are implemented by extending the *Smalltalk* dictionary to include a *layer marker* tag. As this imposes a space and time overhead, a mechanism is provided which allows the context mechanism to be suppressed. As [Goldstein 80b] says,

> "... it remains an important research goal to make the context machinery available to the user in a convenient fashion."

One of the design aims of the *GLIDE2* system[1] [Eastman 80] was to support the exploration of alternative designs. This is done by means of a special form of "checkpoint file", called an *alternative*, which is a temporary file into which the set of database updates are stored instead of overwriting the original values. In effect, this is a form of differential file (see section 3.3 on page 26), though no indication is given as to the way in which these checkpoint files are used by the system. Alternatives may branch from other alternatives, thus forming a tree-like structure, with merging of alternatives being allowed only by a specially privileged user.

[1]Graphical Language for Interactive Design

1.6 Some other uses of such a system

A system which supported experiments would be useful in other fields than CAD. In teaching about databases it is desirable that students are provided with a database which they can use to perform practical exercises, thereby increasing their understanding of the theoretical material presented in lectures. One approach would be to require them to create and populate their own databases. This is not ideal, however, since this requires that the students learn simultaneously how to get information into a database and to get information out. Obviously it would be preferable if they could be provided with a ready populated database for their initial exploratory exercises, and only be required to progress to updating a database once they were confident about querying one. Furthermore, a ready populated database could be set up to contain considerably more information than they would typically supply, and therefore the practical exercises could be that much richer. It would not, however, be desirable, for them to be allowed update access to such a ready populated database, as inevitably, either by accident or design, it would be corrupted. The use of a database system which supported experiments would overcome this last objection, since each student would appear to have his own copy of the database and would be able to modify it independently of any other student.

In exploratory data analysis [Tukey 77] it is often useful to be able to try transformations of the data or to delete outliers in an attempt to make the measurements more amenable to analysis. Such action is directly analogous to the modification of *F* to *F'* in figure 1-2, for example, and a system which implemented experiments would be useful in this context.

1.7 Summary overview

In this Thesis we will be investigating techniques for the efficient implementation of experiments, as motivated in this chapter. In chapter 2 we present an overview of the current state of database technology as it impinges on our work. In chapter 3 (on page 24) we present a number of techniques, with an analysis of how we expect each to perform. Chapter 4 (page 43) presents the results of a preliminary investigation of one of these (differential files [Severance 76]) with chapter 5 (page 49) describing an implementation of a system to support experiments based on the experience gained; the results of an exhaustive sequence of performance tests are presented in chapter 6 (on page 55), together with the result of a comparative trial with the method of shadowing [Astrahan 76, Challis 81]. Chapter 7 (on page 80) considers how best use may be made of a virtual memory environment, listing the operating system requirements for doing so. In chapter 8 (on page 100) we generalise from the single-designer case to the case where a team of designers is working concurrently on a design project. Chapter 9 (on page 108) presents a summary of our work, together with our conclusions, and indicates some directions for possible future research and development. Finally, five appendices describe some supplimentary material: appendix A describes the *Rbase* system [Tate 81] used in chapter 4; appendix B describes the user and operating system interfaces to the implementation of chapter 5; appendix C describes the interface to the *Shrines* system of chapter 7; appendix D briefly summarises the *VAX/VMS* paging mechanism and describes the system parameters which affect it; and appendix E describes how best use could be made of microcode assistance.

Chapter Two
Current Database Technology

2.1 An overview

A *Database* is a structured body of information held in machine readable form in such a way as to be readily accessible and easy to manipulate. Usually the information will be shared between a number of users, possibly large. Often the amount of information stored is quite considerable. Some examples are:

- An airline seat reservation system. A large number of agents are trying to sell seats on a number of aircraft to members of the public. Each agent queries and updates the common database for the system as he finds possible seats for customers, and ultimately sells one. Since it is impossible for two people to occupy the same seat, there must be some way to ensure that a customer is not offered a seat only to discover that it has been sold to someone else before he has time to accept it, or that two customers are not both sold the same seat.

- A stock management system for a manufacturing company. As raw materials, parts and finished products are moved around the company these operations are recorded for use by, say, the ordering department who wish to know when to reorder some component part, the sales department who wish to know whether a manufactured item is in stock or when it is likely to be, and the management who wish to know the pattern of stock movement in order to monitor plant efficiency and anticipate future problems.

- A payroll system. Details of employees of a company would be held in such a system, examples being name, age, sex, department, grade or title, salary, medical records. Some of the information might only be available to certain users of the database, however. For example, salary might only be accessible to the personnel department, medical records to the company doctor, and so on.

- A library system. Such a system would maintain records on the library's collection of books. Information held might include book title, author, ISBN, catalogue number, classification, date of purchase and borrower. Information on accredited borrowers, such as name, address and books on loan would be maintained. In addition the system could allow users to enquire about books given only incomplete information. Examples might be to ask about any books on horses written by an author whose surname resembled "Atkensyn", or to enquire about articles in journals which refer to, say, "density estimation".

- A database storing information about point to point links and terminal lines for the Computer Science department. Such a database would have only a few special users, namely the technicians responsible for maintaining such a network. Information held might include the locations of junction boxes, and for each cable the number of circuits it held and where each circuit terminated. The operations provided by a database system would allow the answers to such questions as how to connect from point A to point B, and what intermediate connections are required. The resulting wiring schedule for each junction box would enable a technician to effect such a link with much less chance of error than if the route had been determined manually.

- A banking system holding information on all the accounts of a bank's customers. These accounts might be queried and updated by bank staff in the various branches of the bank, or directly by the customers via some form of remote facility such as *Cashline*.

2.2 Concurrency control

The possibility of concurrent access to a database introduces many problems of consistency and integrity. The examples above show how the magnitude of these may vary.

At one extreme we have the point to point link database. With this database there will only ever be one user at any time, and therefore there is no possibility of inconsistency introduced by concurrent access.

In the middle of the range is the library system example. The only updates made will be when books are borrowed or returned or when new books are added to the collection. Here the inconsistencies will merely cause a little inconvenience, as, for example, when a potential borrower queries the catalogue and discovers that the book he is looking for is in stock, while at the same time it is in process of being borrowed by someone else. When the potential borrower then goes to the shelves to locate the book he finds that it has gone. The inconvenience caused by this infrequent occurrence is more than compensated for in the general case by the time saved looking for books which are definitely known to be out on loan. Note that as there is only one of each book there can be no inconsistency problems caused by the same book being borrowed or returned simultaneously by several borrowers. Thus in this system the problems caused by concurrency are merely annoying rather than damaging, and will, in any case, be sufficiently infrequent that no special precautions need be taken.

At the other extreme we have the airline reservation system. In this example it is very important that concurrency problems are dealt with properly. An airline would soon lose all its customers if it were a frequent occurrence that seats were sold to more than one customer. The database system must, therefore, prevent such occurrences from happening.

2.2.1 Problems caused by concurrency

Before we consider methods of concurrency control in more detail we must give some more thought to the problems which may be caused by concurrent use of a database. Obviously concurrent reading of the same datum cannot cause any problems. It is not until concurrent reads and writes are attempted that difficulties emerge. Consider the following examples.

- Two users, *A* and *B*, executing sequentially, both read the same item of data, *X*, add 100 to it and then write it back. After both users execute, *X* should have had 200 added to it. The sequence of operations would be:

  A reads *X*
  A adds 100 to *X*
  A writes *X*
  B reads *X*
  B adds 100 to *X*
  B writes *X*

- However, if we allow *A* and *B* to execute concurrently the following may happen:

  A reads *X*
  B reads *X*
  A adds 100 to *X*
  B adds 100 to *X*
  A writes *X*
  B writes *X*

In the first case the effect of *A* and *B* was indeed to add 200 to *X*. In the second case, however, *B* read the original value of *X* before it had been modified by *A*, and hence the final value of *X* after *B* wrote it back was the original value plus 100, not plus 200. The effect of *B* writing the value of *X* is to erase completely all trace of *A*'s update, hence this is known as the "lost update" problem.

- Consider now the following variation wherein *A* adds 100 to two data, *X* and *Y*, and *B* multiplies *X* and *Y* by 2. Assume that *X* and *Y* have the same value. No generality is lost by this assumption, since a similar problem will arise whatever the relative values of *X* and *Y*. If *A* executes followed by *B*, then the values of *X* and *Y* at the end of this sequence will still be identical. However, if we again allow *A* and *B* to execute concurrently the following may happen:

  A reads *X*, adds 100 and writes it back
  B reads *X*, doubles it and writes it back
  B reads *Y*, doubles it and writes it back
  A reads *Y*, adds 100 and writes it back

There are no values of *X* and *Y* for which such a sequence of operations can yield a result where X and Y are equal both before and after this sequence of operations. Hence, although neither A's nor B's updates have been lost, the sequence does not yield the correct result.

Implicit in this last example has been the assumption that A's two operations should be regarded as a single "unit", as should B's. Only under this assumption does it make sense for us to regard the second sequence of operations as "incorrect". This concept of grouping operations into single logical units is very important when considering database systems. Such a logical unit is called a *transaction*. We define a transaction as a (finite) sequence of operations which are performed by the database system in such a way that there is no apparent effect caused by the other (zero or more) users of the database, and which appear to the other users of the database to take effect as a single operation.

Not all patterns of concurrent access give incorrect results, however. Consider again the second example above, but suppose that the order of execution of the individual steps is as follows:

A reads X, adds 100 and writes it back
B reads X, doubles it and writes it back
A reads Y, adds 100 and writes it back
B reads Y, doubles it and writes it back

This order of execution gives rise to the same result as would be the case if transaction A had been completed before transaction B was allowed to start, though the two transactions have been allowed to run concurrently. In general, since transactions can be run in any order, a concurrency scheme can be regarded as "correct" if the transformation it produces in the database can also be arrived at by running the transactions serially in some order. This is reasonable, since each transaction is assumed to transform the database from one consistent state to another, and hence their composition will also transform it from one consistent state to another. This motivates us to make the following definition.

Definition (2.1): Let $T_1, T_2, \ldots, T_n$ be transactions. Then a concurrency scheme C for running these transactions is correct if, for *some* permutation π of $\{1\ldots n\}$,

$$C(T_1, T_2, \ldots, T_n) = T_{\pi(1)} \circ T_{\pi(2)} \circ \ldots \circ T_{\pi(n)}$$

where "o" means composition of transactions. This condition is known as *serialisability*.

Note that, in general, different permutations π will result in different final states for the database. In the example above, running A before B does not give the same result as running B before A. Neither of these concurrency schemes is "more correct" than the other, since A and B are independent transactions. Only if it were desired that one particular transaction should wait for the other to complete would this be the case, but in such a situation they could not be run concurrently.

Each transaction must (by definition) have a definite beginning and a definite end. There are, however, two ways in which a transaction may end. Either it may be *abandoned* for some reason (in the airline seat reservation example on page 9, the customer may have decided that he could not afford to pay the price required for the seat), in which case no modification to the database takes place, or it may be *committed* (the customer handed over his cheque to the agent selling the seat), in which case all modifications to the database take place as if they were performed by a **single** operation.

2.2.2 Locking

The most common technique used to meet the serialisability constraint is *locking*, whereby a user (program) of a database specifies that a particular item of data is not to be made freely available to any other user. The description given below is not intended as a full discussion of locking (see, for example, [Date 79, Gray 79a, Date 83]) but rather as a sufficient introduction that it will be seen why systems which use this technique will have difficulty in meeting the requirements of experiments described in chapter 1. A discussion of locking in practice may be found in [Grimson 80].

We define only the two simplest types of locks. An *exclusive lock*, as its name implies, gives the user (program) exclusive rights to access a datum: no other user may read or modify it. A *shared lock* allows any other user to read the locked datum, but forbids any user, including the owner of the lock, from modifying it. Several users may each have a shared lock on a particular datum, in which case no user may have an exclusive lock on it. Alternatively, one user may have an exclusive lock on a datum, in which case no other user may have a shared or an exclusive lock on it. If a lock is requested on a datum which is incompatible with any locks held by any other users the requesting process waits until the lock may be granted. Thus, the example above could appear as follows:

A requests an exclusive lock on *X*, which is granted
B requests an exclusive lock on *X*, and waits
A requests an exclusive lock on *Y*, which is granted
A reads *X*, adds 100 and writes it back
A reads *Y*, adds 100 and writes it back
A releases its lock on *X*
B's request for an exclusive lock on *X* is granted
A releases its lock on *Y*
B requests an exclusive lock on *Y*, which is granted
B reads *X*, doubles it and writes it back
B reads *Y*, doubles it and writes it back
B releases its locks on *X* and *Y*

The effect achieved here is to force *B* to wait until *A* has finished with *X* and *Y* before it is allowed to run to completion. In this way any inconsistency is prevented.

In the example above the sequence of operations is that *A* and *B* both acquire locks on all the data they are going to access before they attempt to read and/or modify anything. Finally, having performed all the operations they are going to, they release all the locks they have acquired. Such a sequence is not always possible or desirable, however. Consider the case of the payroll system described on page 9. Suppose that all the members of a certain department are to have their salaries increased by 10%. There are two approaches to locking possible here. One approach would be to acquire an exclusive lock on the entire set of personnel, to examine each employee in turn and to award the salary increase if the employee were a member of the department under consideration. Finally, the exclusive lock on the entire set of employees could be released. This approach is not very satisfactory, however, since it dramatically reduces the level of concurrency possible in the system. Alternatively, a shared lock could be acquired to the entire set of personnel, with exclusive locks being acquired on those employees whose salaries are about to be increased. Finally, all locks would be released. This scheme has the advantage that the level of concurrency possible is increased.

Another problem may arise if locks are acquired during the course of the transaction without a discipline being imposed. Consider the following example, where two transactions *M* and *N* both wish to access the same items of data, *V* and *W*:

M acquires a shared lock on *V*
N acquires a shared lock on *W*
M requests an exclusive lock on *W*, and waits
N requests an exclusive lock on *V*, and waits

M cannot proceed until *N* releases its lock on *W*, while *N* cannot proceed until *M* releases its lock on *V*, and hence neither transaction can proceed until the other does. Such a situation is known as a *deadlock*. The example given here is the simplest possible, with two transactions and two items of data. It is possible to construct more complex examples involving more than two transactions and items of data in configurations other than the simple loop shown here. As the number of transactions and items of data involved increases so the difficulty of detecting deadlock situations increases. Deadlock detection and avoidance is discussed in [Gray 79a, Date 83].

Allowing a transaction to convert an already existing lock into a different type can, in some cases, result in an increase in concurrency. Consider the following example where *A* intends modifying *X* eventually, and does not want its value altered in the meantime:

A obtains a shared lock on *X*
.
.
.
A converts its lock on *X* to be exclusive
.
.
.
A releases its lock on *X*

In the interval between *A* acquiring its shared lock on *X* and converting it to be exclusive any other transaction is allowed to read *X*, though none is allowed to write to it, and hence the possibilties for concurrency have been increased.

This scheme is not perfect, and other, more complex, schemes have been proposed to overcome some of its drawbacks. In many cases these depend on exploiting some attribute of the data structure being altered. There may be an optimal granularity of lock which maximises concurrency [Gray 76, Ries 77, Ries 79]. The locks may be on logical statements about the contents of the database rather than on actual entities within it [Eswaran 76]. The database system may decide to grant stronger locks than those actually requested; this over-locking is quite safe, as the desired constraints on concurrent access are a subset of those actually enforced. It is not our purpose to discuss these here, however. We merely note two of the implications of locking schemes which will have a bearing on the implementation of experiments, *viz*

- A shared lock must be acquired on any object if it to be guaranteed that its value will not change during the lifetime of a transaction.

- A transaction must acquire an exclusive lock on an object before it is allowed to alter it in any way. Such a lock is incompatible with any other lock on the object.

2.2.3 Deadlock recovery and rollback

Having discovered that a deadlock exists some steps must be taken to resolve it. This is usually done by *backing off* (or *rolling back*) one of the offending transactions, that is to say forcing it to restart from some earlier point before it acquired the locks which are preventing the other transaction(s) from proceeding. The transaction to be backed off could be chosen as being the one which had "performed least work" in some sense, the one which started earliest (or latest), or simply by choosing one at random. Having rolled back one of the transactions sufficiently that a deadlock will not immediately recur, the transactions are restarted. A similar chain of events would be required if the user were voluntarily to abandon the transaction.

In order that a transaction may be rolled back it is necessary that the original state of the database be preserved, while at the same time allowing the transaction to see its own updates before they are committed. There are two ways of achieving this:

- As each individual update is performed, the previous value of the datum being modified is preserved in a log. The update is then performed *in situ*. If the transaction requires to be rolled back the previous values of any modified data can be restored from the log. This method, known as *before imaging*, is the one usually employed in current database management systems, for the reasons explained below.

- Updates are not performed *in situ*, but rather are placed at some other site in the database with the access paths being modified in such a way that the updated value is found rather than the non-updated value. The transaction is committed either by copying the updated values to the sites of their non-updated originals or by altering pointers so that the new values are found rather than the originals.

It will often be the case that the frequency of update of a database is much less than the frequency of querying. Consider the payroll system example on page 9. Each employee will require to be regularly paid, either monthly in the case of salaried staff or weekly in other cases. The calculation of the wages paid by the company to its employees will require to access each employee in the database in turn. This calculation will access many more objects in the database much more frequently than will operations to update it, such as when an employee joins the company or receives a pay increase. Consequently it would be sensible if the database were arranged to optimise frequent read accesses, particularly sequential read accesses, at the expense of infrequent write accesses.

This can be done by arranging that objects which are frequently accessed in sequence are located physically next to each other on disc. This will result in fewer (slow) disc accesses as several objects may be read into store in one operation. Furthermore, there may be fewer levels of indirection involved than would be the case if objects were not thus arranged, and hence disc traffic may be further reduced. Arranging objects in this way, however, necessarily means that a scheme of *in situ* updates must be used, since otherwise the physical layout of objects on the disc would be disturbed if ever any object were modified. This scheme may be further enhanced by pre-fetching the next sequential object from disc while the current object is being processed; see [Smith 78] for a discussion of this.

Further, the process of abandoning a transaction under a "normal" *in situ* update scheme requires that any modifications made be restored to their previous states as determined from the log. Such a process is relatively expensive, and it would be desirable if it did not have to be performed often. In the kind of use for which such database systems are designed it would not be usual to start a large number of transactions and then deliberately abandon most of them, but rather a transaction would not be started unless it were expected to commit. However, in the CAD context we are describing, a large number of experiments would be started, only one of which would eventually commit, the others deliberately being abandoned. The restoration of the state of the database which this would entail would be an extremely complex task, made all the more so since there could be a substantial set of objects modified by several transactions. Hence an *in situ* method of update would not be practicable.

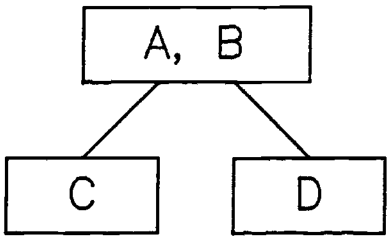
It could be arranged that, with an *in situ* update scheme, any other transaction attempting to read an object which was currently being updated by another transaction would receive an old copy from the log file. Such a scheme would rapidly become unmanageable, however, as there would not be one "old" value, but rather a number of "old" values corresponding to the changes which the various levels of the design tree had made.

2.3 The impact of locking on experiments

It would be impossible to implement experiments using the transaction and locking mechanisms described above, even supposing that transactions could have sub-transactions, a facility not normally provided. The closest approach would be to consider each experiment a "pseudo-user" of the database. Each experiment would open its own transaction on the database, with the designer's changes being made within the context of that transaction. If the designer committed an experiment then the corresponding transaction would be committed, all other transactions being abandoned.

The following example shows how such a scheme breaks down. Suppose we have two experiments open as transactions on a database, as shown in figure 2-1. For convenience we will name the experiments by sufficient of the objects they contain as will uniquely identify them.

Figure 2-1: A simple experiment using transactions



In order that experiment *C* may be guaranteed that the objects *A* and *B* of the main database will be unaltered by experiment *D* it must acquire a shared lock on them, and similarly for experiment *D*. In general, since an experiment will want the entire contents of the main database to be visible and unchanging it will require a shared lock on the entire main database.

Suppose, now, that experiment *D* wishes to alter the value of *A*, to *A'* say. The locking rules insist that *D* acquires an exclusive lock on *A* before modifying it. This exclusive lock cannot, however, be granted while *C* still has a shared lock on *A*, and *D* is forced to wait until such time as *C* releases its lock. This forced suspension of *D* violates one of the requirements of experiments, namely that the designer should not be constrained as to the order in which he suspends and resumes them, and in particular he should be able to resume *D* while *C* also remains open. Hence this scheme does not adequately implement experiments.

2.4 Data models

The first mass-storage medium capable of holding data for computer processing was the magnetic tape. The use of such a medium imposed certain constraints on the method of processing, in that it was essentially a **sequential** medium. Programs were forced to accept their input in a predetermined order and to write their output in much the same order. Access to read individual data, while possible, was slow, and the possibilities of writing individual data and of sharing data were non-existent.

With the advent of improved disc technology came the benefits of fast random access to individual items of data, and a corresponding proliferation of access techniques such as are described in [Dodd 69, Senko 77, Wiederhold 77]. These methods were packaged up as a library of procedures so as to aid the applications programmer by removing the necessity for him to write a large amount of complex code for each new application. Sharing of data was now possible, and as concurrency controls were added to the libraries of access procedures the primordial database management systems gradually evolved. (For a more detailed discussion see, for example, [Fry 76, Sibley 76].)

One large problem with these systems was the intimate relationship between the structure of the data and the programs which accessed them, with the result that altering the storage mechanism, whether to improve efficiency, for example by using access methods more appropriate to the quantity of data, or to add new fields, which were unforeseen when the database was designed, became an expensive undertaking. Furthermore, it is sometimes necessary to restrict the ability of various classes of user to access some of the fields of a database, as, for example, in the payroll system described on page 9. It was therefore found desirable to distinguish between the **logical** database, as seen by a user, and the **physical** database, being the way the logical database is stored by the system, with one or more levels of *schemata* defining the relationships between the various layers of the database and the views which users are allowed to see (this is described in [Date 81], for example). This resulted in the development of the data models described below.

Do-it-yourself databases are far from dead, however, as witness the fact that a number of sophisticated options have been added to the *VAX/VMS* Record Management System [DEC 82a, DEC 82b] since the first release of that operating system.

2.4.1 Hierarchical model

The *hierarchical model* [Tsichritzis 76] is conceptually, perhaps, the simplest of data models, with *IMS* [McGee 77] the most ubiquitous example. In this model the universe is considered to be categorised hierarchically.

Consider, for example, the tertiary education system, which consists, at the topmost level, of several types of institutions, such as Universities, Polytechnics, Technical Colleges, Teacher Training Colleges, etc. Each University consists of a collection of Faculties, each of which consists of a collection of Departments, each of which runs a number of courses, and so on. Clearly, such a structure is better suited for answering queries which follow the database structure than those which do not, and thus there will be some applications for which the model is well suited, but there will be many more for which there are better models.

Although now quite long in the tooth, this model is still in widespread use, with further development taking place [Siwiec 77, Strickland 82].

2.4.2 Network model

The *network model* [Taylor 76, CODASYL 78] permits much more complex interrelationships between the items in a database. In the CODASYL model, items are considered as *Records*, each of which belongs to a corresponding *Record Type*. Records may contain zero or more data items or data aggregates (named occurrences of data items). The interrelationships between records are modelled by *Sets*, each *Set Type* having one owner record type and a number of member record types. Each "set" contains one instance of its owner record type and an arbitrary number of instances of its member record types.

The relationships which may be modelled are essentially one-many. Many-many relations may be introduced into the database by means of a dummy record type which owns two dummy set types each of which has one of the sides of the original many-many relationship as member records.

Consider, for example, a record type *Course*, giving details of courses offered by University departments, and a record type *Student*, which contains details of University students. These two could be related by means of a set type *Attends*, say, with owner-record type *Course* and member-record type *Student*. Then each instance of the set type *Attends* would indicate which students attended a particular course.[1]

Loops may be introduced into the network: suppose that *Student* is related by the set type *IsDirectedBy* to the record type *Staff*, giving details of which students a particular Director of Studies was responsible for; that *Staff* is related to *Department* by a *WorksIn* set type; and that *Department* is related to *Course* by a *Teaches* set type.

In order to relate items in the database it is necessary to use the "sets" to navigate between the record instances. Thus, in our example above we would find the names of all students who attend courses in, say, Computer Science by navigating via the *Teaches* set instances to the *Course* record instances and then via the *Attends* set instances to the *Student* instances.

While a knowledge of the structure of the database is clearly necessary in order to pose a query directly, it may be possible to facilitate operations on the database from the point of view of the user by describing the database in a more convenient form and transforming the queries posed thereon into equivalent queries on the "real" database. In the *ASTRID* system [Gray 79b, Gray 81a] the database is given a relational appearance (see below), with queries being posed in the relational algebra and transformed, after optimisation, into equivalent *RatFOR* program fragments which are linked with some database-specific code and run against the "real" database.

2.4.3 Relational model

Although the network model does provide most of the facilities required in a commercial data-processing environment it is not particularly friendly to the casual user, and indeed quite a fair knowledge of the access paths defined in the schemata is required in order to be able directly to use such a database at all. Furthermore, the model is not particularly amenable to mathematical analysis, although there have been attempts (e.g. [Jacobs 82]), with the result that it is not possible to make soundly-based statements about a network model database. For that reason a number of models have been developed which present both a firm mathematical foundation and a more pleasant user interface. The *relational model* [Codd 70, Chamberlin 76, Kim 79] is one of these.

[1]In fact, since a particular course is attended by a number of students, and each student attends a number of courses, the relationship between students and courses is many-many, and hence a dummy record type would be required to allow them to be linked.

A *relation* is defined as follows:

Definition (2.2): Let $\mathbb{T}$ be the set of all types in the database. Let $\mathbb{D}$ be the corresponding set of domains, i.e. of possible values which objects in the database may take. Let

$$D_1, D_2, \ldots, D_n \in \mathbb{D}$$

Then a *relation*, *R*, over $T_1, T_2, \ldots, T_n$ is a subset of the Cartesian product of the corresponding domains, *viz*

$$R \subseteq D_1 \times D_2 \times \ldots \times D_n \qquad (2.1)$$

Thus, the database appears as a collection of *tables*, each of which has a number of *columns*. The information in the database appears as *tuples* in these tables. The following basic operations are defined over tables:

Select	Tuples may be selected according to the value of a boolean expression over one or more of their fields.
Project	Columns of the relation are "thrown away", and the resulting tuples are formed into a valid relation by eliminating duplicates. Note the analogy of this operation with the usual mathematical operation of projection onto a subspace.
Join	Two tables are joined on a common column by forming the Cartesian product and selecting those tuples whose values in the common columns are identical.

Full-scale implementations of the Relational model include *System-R* [Astrahan 76, Blasgen 77] and *INGRES* [Stonebraker 76, Stonebraker 80, Stonebraker 83]. A comparison of the relational and network models may be found in [Michaels 76].

2.4.4 Functional model

The *functional model* [Shipman 81, Buneman 82] is another model wherein a natural view of the universe is founded upon a solid mathematical base. In this model, objects in the universe, known as *entities*, correspond to points in the domain space of the database, with their interrelationships modelled by functions between the corresponding entities. A full discussion of the functional model may be found in [Kulkarni 83].

Clearly, the functional model and the relational model are closely related. The mathematical definition of a function is that it is a subset of the Cartesian product of its domain and co-domain:

$$f: \mathbb{D} \longmapsto \mathbb{C} \;\equiv\; f \subseteq \mathbb{D} \times \mathbb{C} \qquad (2.2)$$

By taking

$$\mathbb{D} = D_1 \times D_2 \times \ldots \times D_m$$

and

$$\mathbb{C} = D_{m+1} \times D_{m+2} \times \ldots \times D_n$$

in (2.2) and comparing the result with (2.1) on page 20, we see that the functional model and the relational model are syntactically different ways of embodying the same underlying model, *viz* finite set theory.

The two models are distinguished by their philosophy as regards entities and their names: the relational model considers these to be one and the same, while the functional model distinguishes between the two.

2.4.5 Persistent programming

All the above data models were designed with the commercial data-processing world in mind. For CAD, however, it would be desirable if the structures held in the database could more closely model those of the design programs, with the "database system" and the "programming language" being completely integrated [Atkinson 78a, Atkinson 78b]. The programmer is freed from having to worry about details of how to interface to the database system, since this is taken care of automatically, but is instead free to concentrate on the algorithms of the design process at hand. This, of course, is a logical extension of the philosophy whereby the introduction of virtual memory mapped on to disc files (as with *EMAS* [ERCC 78], for example) results in "main" store being regarded not as a separate system component *per se*, but rather as a form of cache for the objects which machine instructions "really" manipulate, *viz* disc files. As [Leesley 79] says:

> "Components of an aircraft or the equipment in a hospital do not have regimented properties and the DBMSs of the banking world are ineffective for handling these data."

Although persistent programming provides what is probably the most useful data model for the CAD environment, the present *PS-algol* implementation [Atkinson 81a] and the proposed *NEPAL* [Atkinson 81b] regard each separate invocation of a program as either a single transaction or a nested sequence of transactions, with all the changes to the persistent data structures being either committed or abandoned when it terminates. This is, at least partly, due to the algorithms used to control the persistent heap: when a pointer is dereferenced and found to reference an object in main store, the object is fetched in from the database and the corresponding pointer patched to point directly to the in-store copy. Thus, having read in an object from one experiment, if the designer suspended it and resumed another experiment any reference to the object would be routed to the wrong incarnation.

One possible solution to this problem would be context-sensitive address translation, analogous to the *PIE* contexts [Goldstein 80b]. We introduce an extra level of indirection between the pointers and the objects they reference. Each entry in the indirection table would point to the head of a list of valid translations for the pointer, each one tagged with the list of experiments to which the translation corresponds. When an object was dereferenced the list of translations would be scanned to determine which, if any, corresponded to the current level. If the translation found was not at the head of the list it would be moved there in order to speed up future searches. This could be speeded further by moving the current experiment indicator to the head of the experiment list. If the corresponding translation was not found then the object would require to be brought in from disc and the appropriate translation performed. Modifying an object would require that a new copy of the object be created and then modified, the current translation for the pointer invalidated for the current experiment and a new translation created. As in the current system, it would be necessary, from time to time, to write out some of the in-store objects to make room for more; indeed, the heap in main store is serving as a sophisticated cache, either for the transaction, as with the current *PS-algol*, or for a number of experiments, as in the scheme suggested here.

Clearly, such a scheme would cause severe performance problems.[1] If we were to arrange that the virtual addresses of objects in the heap remained fixed with the page tables being altered to reflect the selection of a new experiment, however, then we might incur a lesser performance overhead. Essentially, the requirement is that the virtual memory into which the heap is mapped should somehow be made sensitive to the experimental context selected. Such a scheme will be discussed further in chapter 7.

2.5 Summary

In this chapter we have considered the facilities provided by current commercially-oriented database systems, and the techniques which are typically used to implement them. We have seen how concurrent access to a common database may result in loss of integrity. In the past such problems have typically been attacked with some more or less sophisticated locking scheme, such as was described above, in order that transactions can guarantee that the queries and updates they are performing are not in conflict with those of any other transaction. Such transactions are, on the whole, short, and so it is quite reasonable to expect other transactions to wait on those occasions where there are conflicts. Several experiments, all concerned with attacking a design problem with different approaches will, however, require simultaneous read and write access to a common area of the database. As we saw, the required access can not be permitted if a locking protocol is used, and hence that a scheme of *in situ* updates is not practicable. We therefore require to use a scheme whereby a new site in the database is allocated for updated objects. Given such a scheme, concurrency control becomes less of a problem with experiments, since for the single-designer case we know, *a priori*, that only one of the experiments will commit, all the others being abandoned, and hence inconsistencies among them are of no consequence.

Thus features of a database system which are beneficial to a commercial type of application may hinder or even prevent the use of that system for CAD. Similarly, there may be

[1]These could be ameliorated to some extent by making use of microcode assistance. See appendix E (page 144).

applications where the extra overheads introduced to implement experiments would result in unacceptable performance degradation in a commercial environment. It should not surprise us that this is the case, and that a database system designed for one application might be unsuitable for another, as similar situations apply in other areas of Computer Science and elsewhere: we would not expect to write an automatic proof generator in *COBOL* or a commercial data processing package in *ML* [Gordon 79], for example. The fact that the database systems with locking, designed to meet commercial data-processing needs, cannot be used to implement experiments should not be taken as a condemnation of such methods. It merely shows that they are unable to do something for which they were not designed and were never intended.

Chapter Three
Implementing Experiments

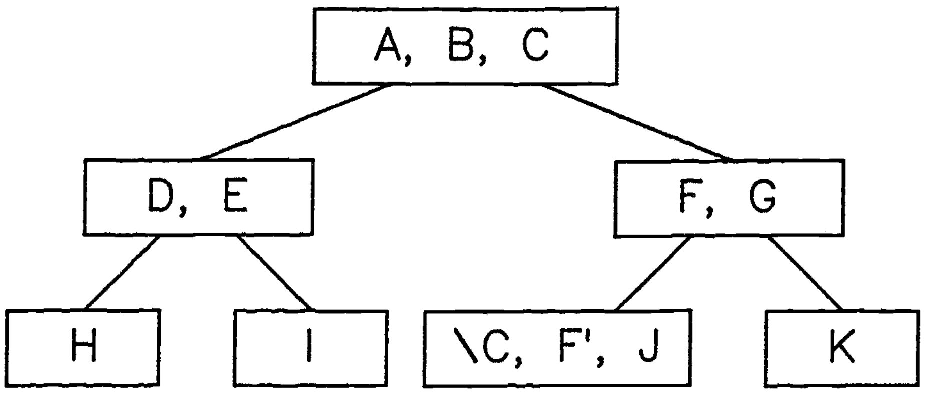
As we mentioned in chapter 2, the alternative to *in situ* updates is to write the new value of an object to a new site in the database and arrange that whenever the updating transaction accesses the object it receives the new copy, all other transactions receiving the old copy. A transaction may be abandoned simply by forgetting any new values it may have created, returning the space released to the common resource pool. A transaction commits by having its updates incorporated into the main database. Thus each transaction has, in effect, its own private copy of the database to modify as it wishes. The extension of the scheme to experiments is immediate: we merely need to allow multiple layers of updates and naming of nodes.

Given such a scheme, alternatives to locking may also be considered. In the case of a single designer working on only one aspect of a design at any time, it may be the case that only a very crude scheme is required. Each experiment in the design tree has, effectively, its own copy of the database. Only one of the experiments is going to commit, the others being abandoned, since there can only be one final version of the design. Thus, each experiment can modify its own copy of the database, quite independently of all other experiments, safe in the knowledge that any changes it makes will either be correctly reproduced in the main database should it commit, or will not be incorrectly reproduced should it be abandoned.

Consider again the example of a design tree given in figure 1-2, reproduced here as figure 3-1. The update "\C" in experiment *J* is not visible to any other experiment, for example. Furthermore experiment *J* can never interfere with any other experiment since either it will commit, in which case all the other experiments will be abandoned and therefore will not exist to be interfered with, or some other experiment will commit, in which case experiment *J* will be abandoned and so will be unable to interfere with the experiment which does commit. Hence, provided each experiment has its own effective copy of the database, no concurrency control scheme of any kind is required, whether using locking or any other method.

The essential difference between this example and the examples on locking in chapter 2 is that here no updates can be interfered with by experiments concurrently modifying the same objects, since all except one of these experiments will be abandoned, whereas in the locking examples (section 2.2.2 on page 13) the updates performed by both transactions were

Figure 3-1: Example design tree



required to be manifest in the database since both transactions were expected to commit, and hence some scheme was required to prevent mutual interference.

Clearly, complications will arise if the designer decides to suspend work on one part of the design and turns to another part, since there may be areas of overlap with part of the database being updated by both sets of experiments. This situation is, however, effectively the same as that of several designers working concurrently on (nearly) disjoint parts of the design, which we will discuss in chapter 8; we concentrate here on the simpler case of one designer working on a single aspect of a design.

We saw in chapter 1 that each experiment should be able to read from and write to the database unaffected by the behaviour of any other experiments which may be accessing the same database. This was shown in chapter 2 to be, at the very least, an extremely difficult goal to achieve using classical techniques. In this chapter we consider alternative implementation techniques, comparing their properties in order that we may decide which is most suited to particular combinations of circumstances. In essence, the requirement is that each experiment should appear to have its own private copy of the database.

3.1 Database copying

The most obvious way to give each experiment its own **effective** copy of the database is to give each its own **actual** copy. Thus, when an experiment opens, the entire parent database is copied, any updates then being made to this private copy. If the experiment is abandoned its database is deleted; if it commits its parent's database is deleted together with all its other dependents, since the committing of an experiment implies the abandoning of all its siblings and their offspring, and its own database becomes the definitive one.

This scheme is conceptually very simple. It corresponds to most systems' views of text editing. In effect it is how systems such as *ESDL* [Smith 80a, Smith 80b], *ILAP/CIF* [Hughes 82] and *Scale* [Buchanan 82, Marshall 83] are used at present, each new experiment corresponding with a new version of the program source. Abandoning an experiment corresponds to deleting a version of the source. Committing corresponds to purging all other versions.

Clearly, however, this scheme has several drawbacks. If the database is at all large then the overhead required to open an experiment, namely copying the entire database, may be unacceptable, particularly if only a small change to the database is envisaged. Further, the method is inefficient in its use of disc space: at any particular time there may be several dozen almost identical copies of the same database in existence, and indeed the user may have insufficient filespace allocation left or the discs may be so full that making such a copy of the database is either forbidden by the operating system or physically impossible. It would be beneficial if some way of sharing those parts of the database which were identical could be found, while at the same time keeping separate those parts which have been changed.

3.2 Repeated editing

If, instead of writing out the entire file each time, our text editor writes out a journal file indicating, in some canonical form, what changes have been made, then we will have a means of reproducing the results of any of our edits with a great saving in space [Zhintelis 79]. While this method may result in the most compact representation of a collection of designs, it does so by trading off space against execution time. The reapplying of all the changes made in an experiment each time it was selected would prove to be quite a large overhead, violating one of our requirements, *viz* that it should be easy for the designer to switch between experiments at will. For that reason we will not consider this approach further since, as we will see, there are other possible ways of implementing experiments without incurring such overheads.

3.3 Differential files

Efficiency of space usage can be improved by making use of *differential files* [Severance 76]. Each virtual file consists of two parts, a read-only *static* part and a writeable *differential* part which contains the changes made to the database. Each of these parts of the database is kept in a separate file in the host filing system. When an object is modified, the new value is written to the differential part and pointers are set to indicate its location therein. When reading, the differential part is first checked to find whether the object has been modified; if it has been, the value from the differential part is used since this is more recent than that of the static part. If the object has not been modified it will not be in the differential part, and so the value from the static part is used. The algorithm used is shown in figures 3-2 and 3-3 on pages 27 and 28.[1] Note that this corresponds exactly with the requirements of experiments described in chapter 1: if an object has been locally modified in

[1]The sections in dotted boxes in these figures will be explained in section 3.3.1 below; they represent refinements to the basic algorithm which can be ignored for the present.

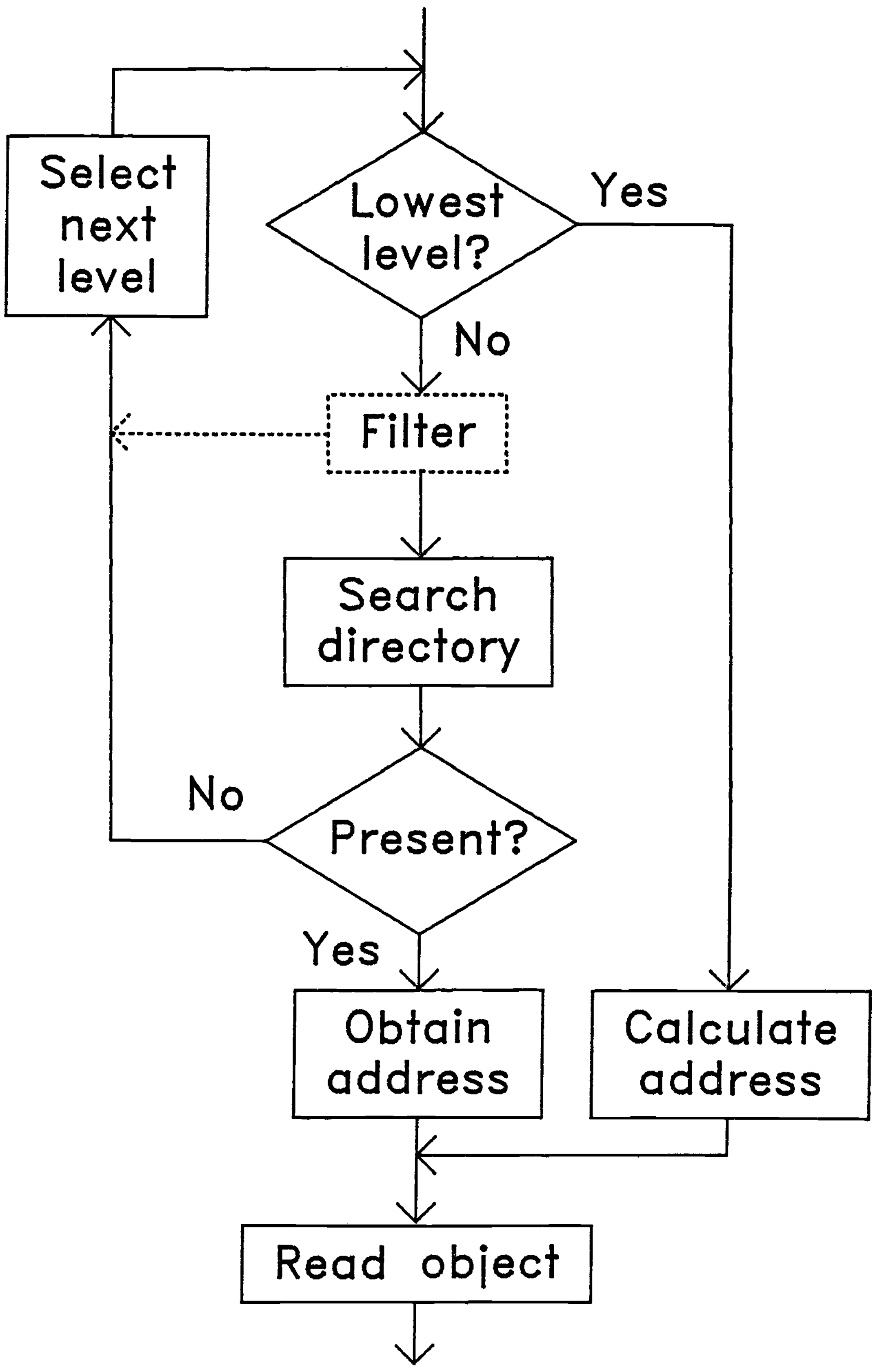


Figure 3–2: Differential file access (reading)

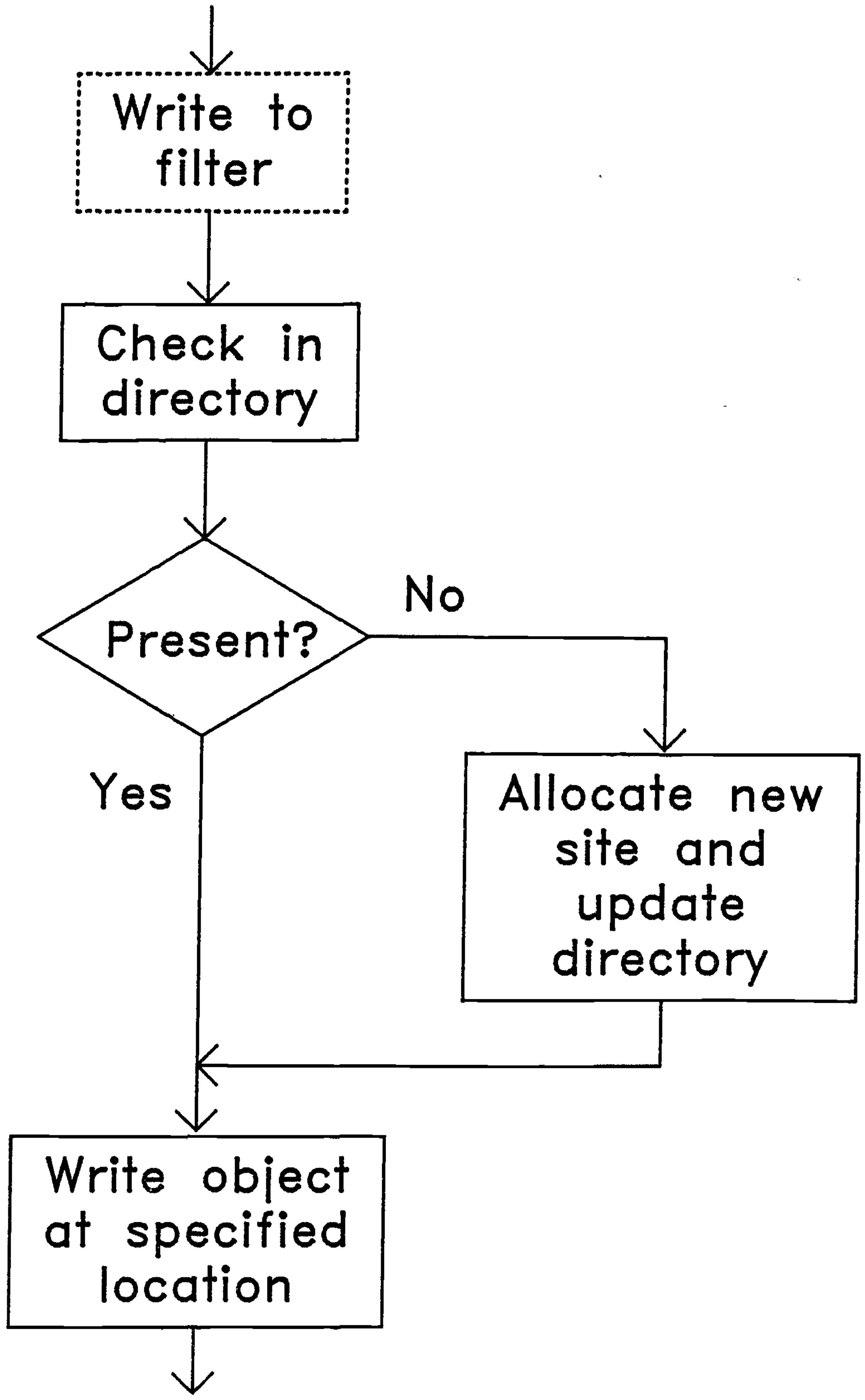

Write to filter
Check in directory
Present?
No
Yes
Allocate new site and update directory
Write object at specified location

Figure 3–3: Differential file access (writing)

an experiment then its new value should take precedence over any value it may have had in the experiment's parent.

3.3.1 Bloom filters

The necessary check to determine whether the object is in the differential part may require several disc accesses and quite a large amount of computation, since the object directory of the differential part may be quite large, and hence there may be a reduction in efficiency of access to objects. This check can be speeded up, however, if a *Bloom filter* [Bloom 70] is included in the differential part. The purpose of the Bloom filter is to provide some indication as to whether or not the object is likely to be in the differential part; if the indication is that the object is definitely **not** in the differential part then the costly process of checking the address translation tables may be avoided. However it is quite allowable for the filter wrongly to indicate that the object **is** in the differential part, provided that it does not do so "too often", since the error will be discovered when the translation tables are accessed. There will still be an overall saving if the reduction in cost caused by not having to access the translation tables for objects which are not there is greater than the extra cost incurred by using the filter.[1]

This requirement can be met using a bit-table and hashing functions. When an object is first written to the differential part, an index into the bit-table is calculated using a hashing function and the corresponding bit set. When an object is read, the bit-table is again hashed into and the corresponding bit checked. If this bit is clear then the object is **definitely not** in the differential part. If it is set then the object may be in the differential part and the directory is searched to determine where the object resides in the differential part, if indeed it does, or whether the bit being set is in fact a false positive. The bit-table will normally be sufficiently small that it can be cached in main memory. The algorithm is summarised in figures 3-4 and 3-5 on pages 30 and 31.

False positives arise since several objects may hash onto the same bit. Indeed the bit-table need not be large enough that each object can be allocated its own unique bit, as would normally be the case for bitmaps. Since all that can happen on a false positive is that the object directory is searched unnecessarily, the only consequence of the non-uniqueness of objects' indices into the bit-table is a loss of efficiency. Nevertheless, if the hashing function and the size of the bit-table are such that false positives are "acceptably" infrequent there will still be a net gain in access efficiency to the differential part, since the extra overhead incurred by hashing into the bit-table will be more than compensated for by the reduction in disc traffic obtained by the elimination of unnecessary directory operations.

The incidence rate of false positives can be reduced by using two or more independent hashing functions into the bit-table. On a write operation, all the bits addressed are set.

[1]This operation could be further speeded by making use of in-line machine code instructions or microcode assistance. See appendix E (page 144).

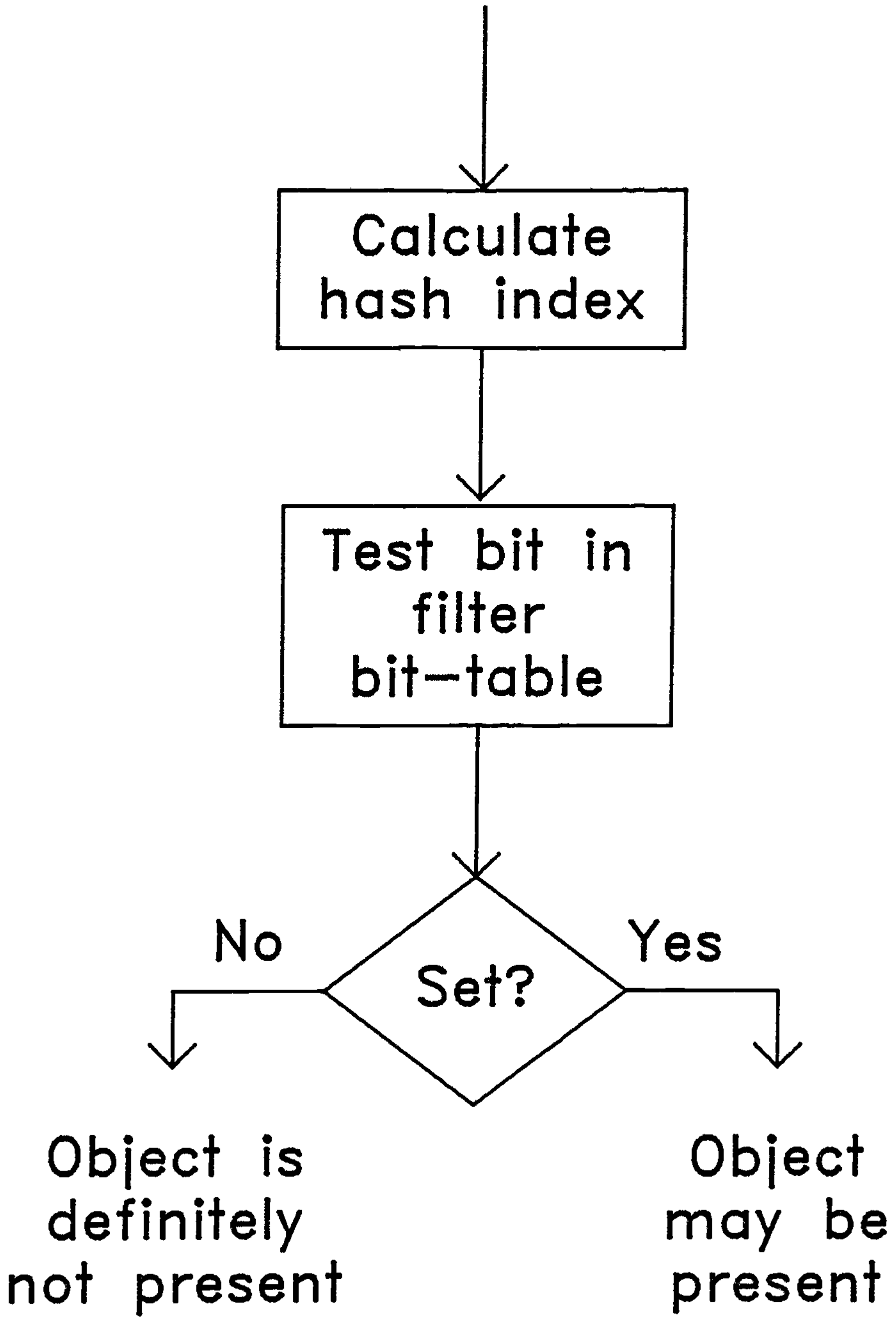

Calculate hash index
Test bit in filter bit-table
Set?
No
Yes
Object is definitely not present
Object may be present

Figure 3–4: Bloom filter algorithm (read)

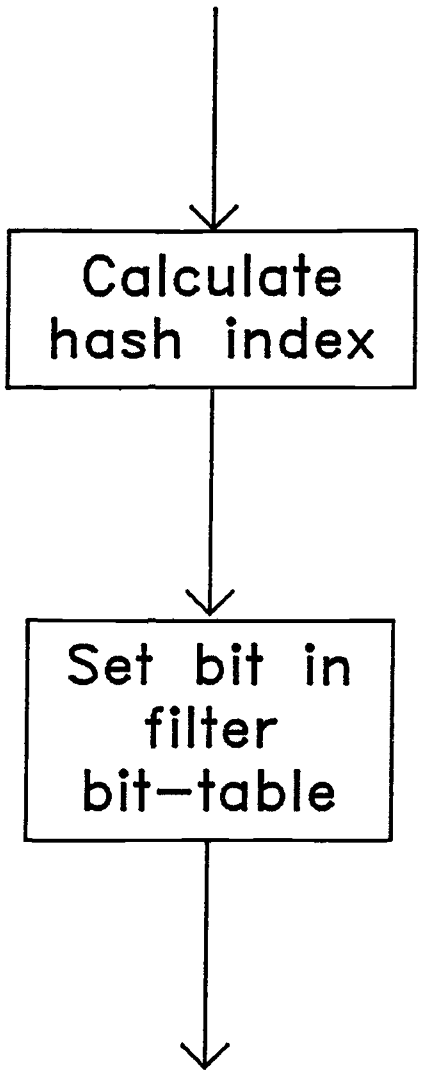


Figure 3–5: Bloom filter algorithm (write)

On a read all the bits indexed are checked: if **any** of them is unset then the object is not in the differential part, while if **all** of them are set then the object **may** be in the differential part. If the index functions are carefully chosen then two objects for which the results of one of the hash functions are identical will probably have different hash values for the other functions, and hence the presence of one of these objects will be less likely to give false positive indications for the other object.

There is, of course, a break-even point where the extra computation caused by adding another hashing function will outweigh any benefits which might accrue. Furthermore, the various hashing functions cannot operate independently of each other since they are all used to access the same bit-table. Adding another hashing function will result in the pattern of set bits in the bit-table being less sparse, and hence make it more likely that some of the bits to which a particular object hashes will have been set as a result of one of the hash functions of some other object yielding the same bit. An optimal choice of hashing functions depends on the pattern of updates in the database, and is dependent on the particular application [Severance 76, Gremillion 82], although it might be possible in some circumstances to find a non-optimal choice which nevertheless gives a reasonable performance over a range of conditions. The system described in chapter 5 uses only a single hashing function; as will be seen, this gave quite acceptable results.

3.3.2 Committing and abandoning with differential files

As with the copying scheme described above, abandoning an experiment is quite straightforward: the differential part is simply deleted. Committing an experiment requires that the differential part be merged with the static part. This can be done by sequentially scanning the object directory for the differential part and updating the static part *in situ* to have the new values. Only when this is complete can the differential part be deleted. It is important to note that a system crash while the merge is taking place will not result in loss or corruption of data: the merge is merely restarted when normal service is restored. There is no need to determine how much of the merge took place in order to restart from that point since there is no loss (except in time) in restarting from the beginning. Objects which were successfully updated before the crash took place are merely updated again, and those which were not are updated for the first time. Thus the method is robust against such mischances provided the differential part is kept until such time as the integrity of the merged static part is assured. Furthermore, there is no need to restrict access to the database while the merge is taking place if the system redirects accesses to proceed via the differential part until such time as the merge completes, since the view of the merged database and the view by way of the differential file are logically identical, and hence committing may appear to the user to be virtually free.

3.3.3 Other advantages of differential files

This differential file scheme has other advantages over *in situ* updating and database copying. Experiments may last for a considerable time in the case of a complex design, and in particular may overlap with one or more of the regular backups of the system filestore.

Since the static part is read-only, it will only require to be backed up once. Any updates made since the last backup will appear in the differential part, which will be smaller than the static part, probably substantially so, and will therefore be quicker and easier to back up. In the case of *in situ* updates, the main database is **not** read-only and will therefore require to be completely backed up. In the case of database copying, the main database **is** unchanging (it is never accessed once the copy is complete) and so only requires backed up once, but the copy owned by the experiment will be very similar in size to the original owned by its parent, and so no benefit accrues.

A further advantage enjoyed by both the differential file method and the database copying method is that since the main part of the database is never written to, it may be protected against accidental corruption, either by physically or logically write-protecting the disc on which it resides or by setting explicit file protections against writing. As the database will contain the most up-to-date copy of the design, possibly even the only machine-readable copy, it is a valuable structure. The extra safety which this additional level of protection affords may prevent loss or corruption of the database, with the consequent waste of investment which results.

Differential files were originally proposed [Severance 76] as an aid to the maintenance of large databases. The advantages envisaged included recovery [Verhofstad 78, Bhargava 81], reduction in dump times, and on-line reorganisation.

3.4 File copying and differential files compared

Of these two methods, the differential file scheme appears much more attractive than database copying, except perhaps for very small databases. Database copying carries a large overhead when an experiment starts. Differential files carry a lesser overhead when an experiment commits, since only those objects which have been modified require to be copied, a much smaller overhead since the update density will usually be low; in any case, the commit appears to the user to cost very little, since he may proceed with using the newly-committed updates immediately, the actual merge taking place as a background task. Even with the comparatively high update density of the test runs described in chapter 6, the cost of merging an experiment using differential files was found to be less than half the cost of opening just one experiment using file copying (see section 6.3.1 on page 60). Furthermore, with database copying the overhead is incurred for **all** experiments, whether or not they will eventually commit, whereas for differential files the copying overhead is incurred **only** for the transaction which commits, and not for those which are abandoned. The differential file method does carry the extra overhead of requiring that the bit-table and possibly the object directory require searching for each access, but with careful design this can be kept acceptably low. Finally, database copying is a much less efficient user of disc space, and this fact alone may effectively forbid its use.

3. 5 Shadowing

It will not always be the case that there is a host filing system available for use by a differential files system: for example, a stand-alone system running on a bare machine would have direct control over the discs attached; or it may be necessary from an efficiency point of view to bypass the host file system interface to the discs and access them directly [Stonebraker 80, Stonebraker 81]. *Shadowing* [Astrahan 76, Lorie 77, Challis 78, Harder 79, Challis 81, Gray 81b] is a technique which can be used in such situations. In this method the entire database is maintained as a single unit, partitioned into blocks of equal sizes, not necessarily identical to a disc block. All blocks are accessed indirectly via a directory, known as the *Logical to Physical map* (LP-map), in a similar way to objects in the differential part using the differential file method. Each experiment on the database owns its own copy of the LP-map. When an experiment updates a block, it is not written back to its original site, but rather a new site is claimed from the common pool of free sites in the database, the block is written to that new site, and the experiment's copy of the LP-map is updated to show the new location of the block. If an experiment commits, its copy of the LP-map becomes the definitive one and the old sites of any blocks it has modified are returned to the free site pool. If an experiment is abandoned, the sites of any blocks it has modified are returned to the free site pool and its copy of the LP-map is forgotten. The method is made robust against system failures by ensuring that the old LP-map on disc is not invalidated before the new one has been safely written out.

With this basic algorithm the LP-map must be sufficiently large that it can accommodate the largest required database, with the result that there is an unacceptably high overhead incurred for smaller databases. This can be ameliorated, however, by using a multi-layer LP-map and partitioning the block addresses. With this scheme only those parts of the map which are actually used need be present, with all others assumed to hold null entries.

This is the scheme used in the *Shrines* utility described in section 7.2 and appendix C. Here the basic blocks are 512 bytes in size; the shrine must not be larger than 64000 blocks. There is a two-level LP-map: at the top level there are two *root blocks* which are used alternately to implement transactions using a sequence number. When a virtual file is opened the root blocks are examined and the one with the higher sequence number is taken to be the more recent. When the virtual file is closed the sequence number is incremented and the root block is written back to the site of the other root block, thus preserving two generations of database state. That part of the root block not taken up by the sequence number and information about the size of the shrine is given over to the first-level map, consisting of 200 16-bit unsigned entries. Each entry in this points to a second-level map page consisting of 256 16-bit unsigned pointers to the data pages. The top 8 bits of the (16-bit) virtual block address are used as an index into the top-level map, with the bottom 8 bits used to index into the resulting map page. An example of this mapping for a particular virtual address is shown in figure 3-6 on page 35, with the algorithms used to access the structures for reading and writing summarised in figures 3-7 and 3-8 on pages 36 and 37.

The free block list is maintained as a bitmap in this implementation: when a transaction is opened the entire LP-map is scanned, with any block not appearing being assumed to be free and marked as such in the bitmap; when a transaction is closed (or abandoned) any unused

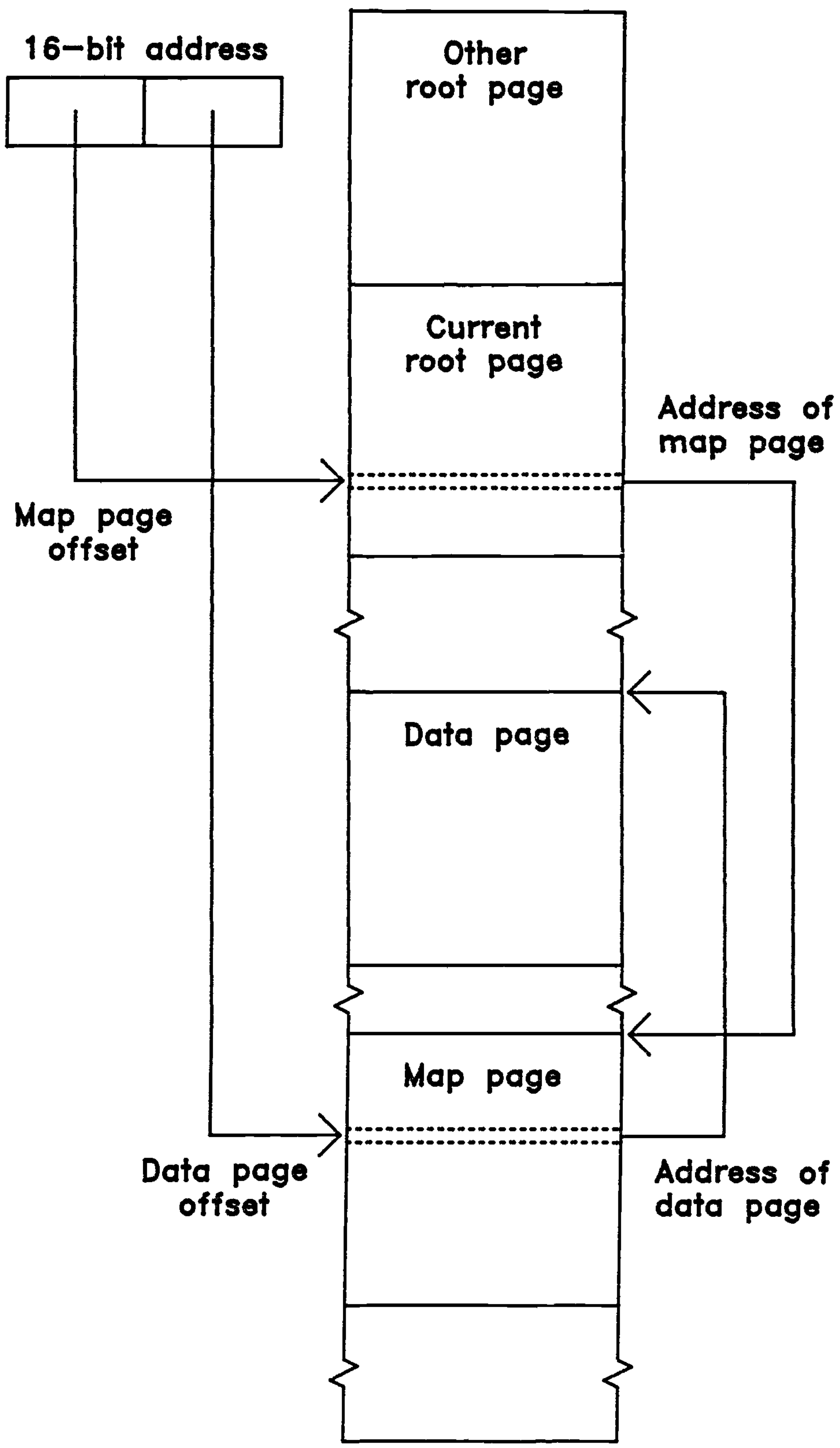


Figure 3-6: Two level map

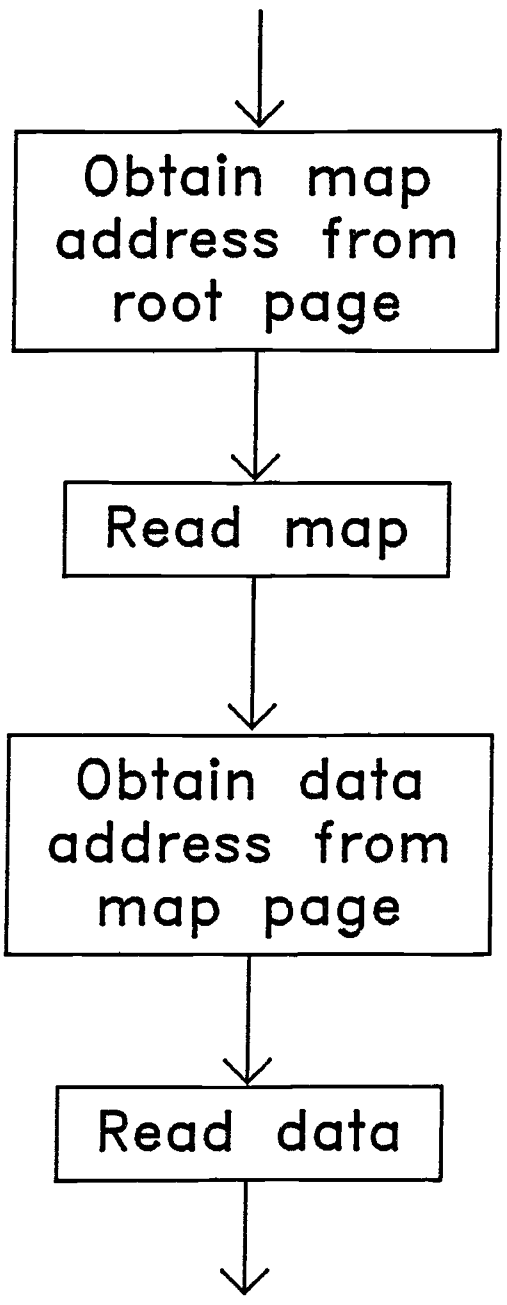

Obtain map address from root page
Read map
Obtain data address from map page
Read data

Figure 3–7: Shadowing (read)

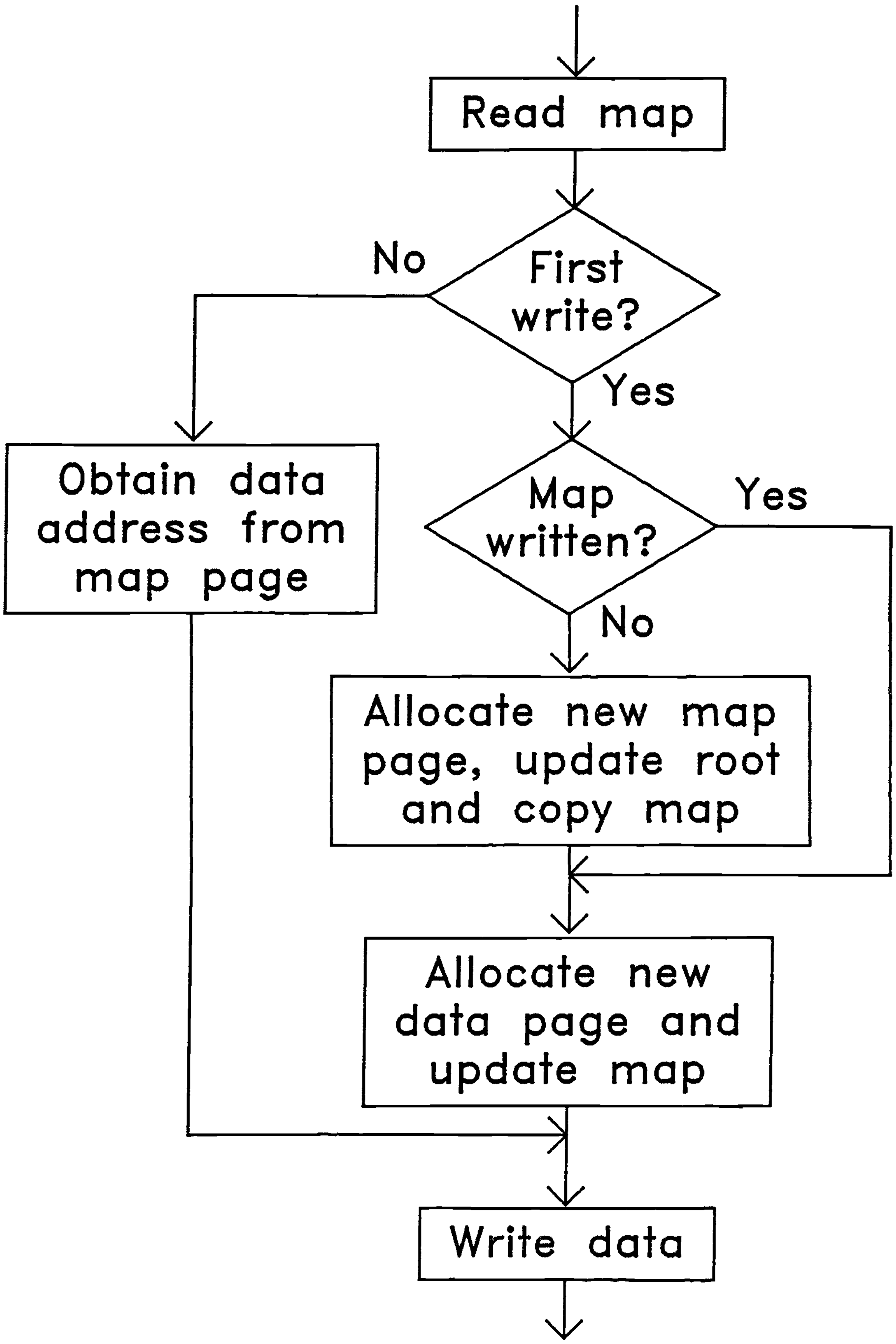
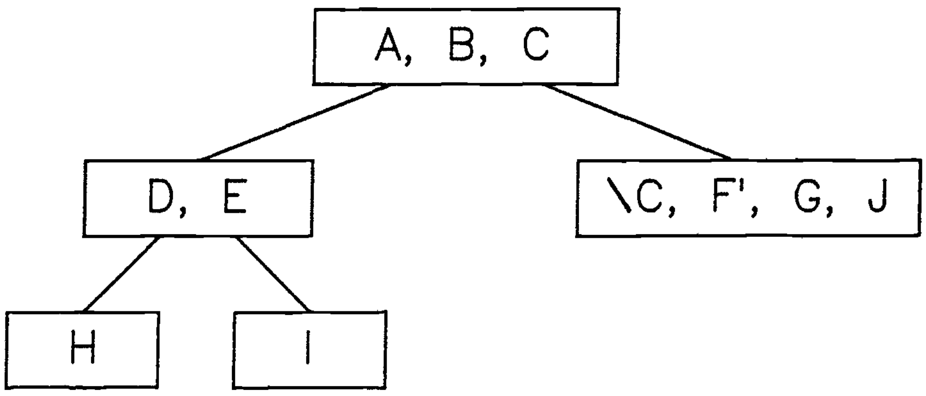

Read map
First write?
No
Yes
Obtain data address from map page
Map written?
Yes
No
Allocate new map page, update root and copy map
Allocate new data page and update map
Write data

Figure 3–8: *Shadowing* (*write*)

blocks automatically become free by virtue of the fact that they will not appear in the LP-map when the virtual file is next opened. An alternative scheme would implement the free block pool as an explicit free list, with blocks being allocated as from it as required and deallocated to it when the transaction was closed or abandoned. This latter scheme, while avoiding the starting-up overhead of building the bitmap, requires a more complicated closing-down algorithm, since the LP-maps must be compared in order that blocks which were used in the old map but are no longer used in the current map are deallocated to the free list.

Note that the scheme described here implements **transactions** on virtual files, **not** experiments. The latter would require that sites exist in the disc file for the root blocks of all the currently active experiments. Furthermore, committing an experiment would require that the LP-map of the experiment's parent be compared against **all** the other LP-maps in order that blocks which were no longer used could be deallocated to the free list, since as there may be several experiments active simultaneously there is no guarantee that a block which was used in the former parent experiment but is no longer required in the newly-committed experiment is not still required by some other experiment: in figure 1-3 (reproduced as figure 3-9), for example, it might be the case that the block containing *C* is no longer required in the right-hand branch, since \C has superseded it, but it cannot be released to the free block pool since it is still required by the experiments on the left-hand branch of the tree.

Figure 3-9: Example design tree



3.6 Shadowing and differential files compared

It will be seen that shadowing and differential files have much in common: both make use of indirection to perform updates, writing new values to a new site rather than back to their original site. There is, however, an important logical distinction between the two approaches: with a differential file the unchanged state of the database is held separately from the updates; while with shadowing the unchanged state and the updates are part of the same structure.

As we will see, this logical difference gives rise to a number of practical differences. Let us consider the relative overheads of the two methods, using as a basis for our calculations a block-based system. We assume that the unit of "differentiation" is the disc block. This is reasonable, since higher-level software will arrange that related records will be grouped together in the same block (or possibly *frame* [March 81], constructed out of a number of blocks) for reasons of efficiency of access; these related records will tend to be updated together, and hence although we may carry some unmodified records into the newly-allocated block, the space we lose is balanced by a saving in translation table size and a gain in access efficiency.

For both schemes the costs of opening experiments are comparably low: with differential files all that is required is that a new file is created and its administrative area initialised; with shadowing the (top-level) map is copied. Neither of these operations will incur great overheads. Abandoning and committing are also both cheap with differential files: to abandon an experiment all that is required is that the differential file be deleted, while committing merely requires a decision to route all subsequent accesses to the database via the newly-committed experiment. Clearly, efficiency would dictate that the committed experiment should be merged into its parent, but there is no reason why this should impede access to the database: accesses to blocks which were not modified by the experiment will reach the former parent as before; while blocks which the experiment did modify will come from the differential file until such time as the merge has completed, thereafter coming from the merged file. The merge may take place at a time which is most convenient for the system, possibly overnight. By similar reasoning, the garbage-collection required after committing or abandoning using shadowing could also be postponed until the most convenient time, the only operation being necessary at the time being to ensure that all subsequently opening experiments should copy the newly committed LP-map rather than any previous one.

Using the differential file method, there is virtually no translation required at the base level between virtual block number in the database and logical block number in the file which holds the database: the most that will be required will be the addition of a constant to take account of header and administrative blocks at the start of the file. The logical block number is translated by the filing system to the physical block number of the block on disc. This is presented to the I/O system which will read in the required block. Using shadowing, the virtual block number is translated, via the LP-map, into the physical block number which is presented to the filing system as before (or the I/O system, if the disc is dedicated).

For differential files above the base level, the work required to translate the virtual block number to the corresponding logical block number is greater than that required for the base part, as the object directory must now be searched, whereas for the base part only a constant need be added. The logical to physical translation, performed by the filing system, then takes place. For shadowing, there is still only one translation required, from virtual to physical. In effect the single translation performed for shadowing is equivalent to the composition of the two translations performed for differential files.

Let us consider further the translations involved in each case. The logical to physical translation performed by the filing system will typically involve scanning a (small) table, to determine in which extent of the file the logical block resides, and then adding in the appropriate offset from the start address of the extent on disc to determine the physical address. The organisation of the filing system will have been arranged so as to keep files in a small number of large extents rather than a large number of small extents in order to minimise the overheads incurred in scanning the extent list, and to keep the size of the extent lists as small as possible so as to allow the filing system to cache as many as possible in main store to minimise disc traffic. The differential file system will arrange to extend its files so as to aid this process. The work involved in the logical to physical translation will, therefore, be correspondingly small, possibly no more than half a dozen in-store comparisons followed by a subtraction and an addition, and hence the overheads involved in accessing a disc block via the filing system as is required for the differential file method, are not great.

This is the case with the *Mouses* Director [Culloch 79, Culloch 82] where each file is described by a *spine block*. This contains 8 bytes of header information (access rights, a checksum, the total size of the file) and 126 4-byte entries. Each of these entries describes one of the disc extents containing the file, giving its size and its physical starting address. Where the discs are regularly compressed as part of the backup operation it is typically the case that even large files occupy only a small fraction of the extent list, with the result that the overheads incurred are negligible compared to those due to disc latency. Even with a badly fragmented disc, the overhead of searching the entire extent list would compare very favourably with the head seek time which would almost certainly follow.

Shadowing requires that every virtual block address be translated via the LP-map to the corresponding physical block address. It is no longer possible, however, to use a simple extent list, as in a filing system, since consecutive logical blocks need not occupy consecutive physical blocks; instead some form of lookup table will be required. This table must contain an entry for every virtual block in the file, however, rather than for only a few significant virtual addresses, with the result that the structure of the LP-map is very much larger than that of the spine block, which was exactly one disc block in size. This extra size will mean that it will be more difficult for the LP-map to be made resident, with a resulting increase in disc traffic and a corresponding decrease in throughput. Furthermore, internal fragmentation will mean that consecutive logical blocks will not necessarily be physically adjacent, with the result that performance will suffer in comparison to a system which can choose how best to place blocks.[1]

The following example will serve to indicate the disadvantage incurred. Suppose each entry in an extent list consists of two 32-bit quantities, namely the logical start address of the extent in the file and the physical start address of the extent on the disc. Extent entries are arranged in the extent list in increasing order of logical start address. The extent list occupies a single 512 byte disc block. One of the entries must be a dummy to indicate the size of the file. Each extent list entry is eight bytes long, and therefore 63 entries can be accommodated in a single disc block. If the differential file system extends its files in contiguous units of 64 blocks each, then the extent list can, in the worst case where the filing system is unable to merge adjacent extents, address 4032 blocks, i.e. 2 Mbyte. In practice, unless the filing system is heavily congested it will be able to merge adjacent

[1]See also [Smith 78].

extents, and as a last resort a disc compression will convert all files into single extents, with the result that the largest file which can be addressed from an extent list held in a single block is the entire disc.

In contrast, each virtual block accessed via a LP-map will require its own entry in that map. Each entry will require to contain the physical disc address corresponding to its index in the LP-map. This will be 4 bytes in size, and hence each block will be able to accommodate 128 entries. To address the 4032 blocks addressable via an extent list as above would require 32 blocks. Disc compression cannot help us to address a larger area in this case.

A similar translation is required for each block in a differential file, as here again logically adjacent blocks are not necessarily physically adjacent. In this case, however, each translation entry would consist of two 4-byte quantities, and hence 64 entries could be accommodated per block.

Hence to address a 4032 block main database with a 64 block experiment using differential files will require one extent list for each of the two files to be cached by the filing system, and one block of virtual to logical translation tables and a one block bitmap (= 4096 bits) to be cached by the database system, giving a total of four blocks to be cached, a sufficiently small number that this is practicable. The corresponding size for shadowing would be 33 blocks, assuming that all 64 new translations are from the same block of the LP-map.

Overall, then, we would expect that a system using differential files would incur lower overheads than one using shadowing, due to the smaller space requirements of its address translation with the corresponding reduction in disc traffic which results.

Finally, differential files allow quicker backup and the ability to write-protect the static part of the database, as described above. Both these very important operational advantages are denied to systems using shadowing since the database is maintained as a single monolithic unit with no internal structure visible to the operating system.

3.7 Summary

In this chapter we have considered four methods of implementing experiments, *viz* file copying, repeated editing, differential files, and shadowing.

Although the simplest approach, and the one which most closely resembles the conceptual requirements of providing each experiment with its own private copy of the database, the *file copying* approach has large space overheads and a startup cost which increases linearly with the size of the database. Thus, it is only really practicable for very small databases, of the order of a few kilobytes.

With *repeated editing* a canonical form of instructions is stored, which allow the updates to be repeated as and when required, rather than storing the actual results of performing the updates. While permitting a very compact representation of some forms of large-scale update, this method has the disadvantage that a high processing price must be payed each time a new experiment is selected, as all the updates which define it must be reapplied.

Both the *differential files* method and the *shadowing* method operate by redirecting modified blocks to new sites on disc rather than back to their old ones. The difference between the two approaches is that with shadowing the new blocks are part of the same monolithic structure as holds the unchanged part of the database, whereas with differential files the changed and unchanged parts are kept clearly distinct. This gives a number of advantages to the differential files approach: the address translation schemes may be separately optimised for the static and differential parts (in particular at the lowest level the translation will be linear, and hence no translation table will be required at all), whereas with shadowing the same translation algorithm must be employed throughout; since the static part is unchanging, any incremental backup of the host filing system will see a much smaller amount of changed data, whereas with shadowing the entire file holding the database will be deemed by the host filing system to have changed and so will require to be backed up; and since the static part is kept distinct from the differential part it may be write-protected, either logically or physically.

Chapters 4 to 6 describe implementations of experiments by means of differential files, with section 6.12 on page 78 reporting the results of a set of measurements comparing the relative performance against a transaction system based on shadowing.

Chapter Four
Differential Files - a Preliminary Investigation

4.1 The Implementation

The first implementation of the differential files technique undertaken was intended not as a full implementation of experiments but rather as a pilot study to establish that the method did, in fact, work in practice, and to help identify any practical problems which might arise. Neither was it intended that the tests performed should reflect CAD usage of a database system; the sole reason for the choice of queries was that it was necessary to load the database with a reasonable amount of structured data in order to exercise the differential file system.

The implementation was not performed "from scratch", but as an extension to the already existing database system *Rbase* [Tate 81], originally written to run under the Edinburgh Multi-Access System (*EMAS*) [ERCC 78, ERCC 82] using two or more mapped files to hold its data structures. The first of these files is a header file containing a dictionary, information on allocation of space and pointers to subsidiary files containing the actual data in the database. As the database becomes full, additional subsidiary files may be added to increase its size. The maximum number and the sizes of the subsidiary files is set at compile time, the value chosen when the system was ported to the Computer Science Department's *VAX-11/780* being ten files of 200 blocks (100 Kbytes) each.

The data model supported is a semantic net: this provides features of both the relational and the network model, and also of Artificial Intelligence languages such as *CONNIVER* [Sussman 72]. Items in the database consist of tuples of order from one to five. Each element in a tuple may itself be a tuple. The outermost tuple may also have a value attached to it. In addition to these relation-like features there are facilities for accessing sub-tuples as tuples in their own right and for traversing the network structure used internally. The user interface to the query program used is described in Appendix A.

For the implementation using differential files, the header and subsidiary files were merged into one, simplifying addressing but preventing the growth of a database by the accretion of more subsidiary files. As the database used in the tests was of known size, however, this caused no problems in practice. The major implementation difficulty was caused by the fact that although the data structures were constructed in contiguous areas of virtual store, these were invariably accessed as individual words and not as part of any larger structure. In order to add differential files to the system it was therefore necessary to replace the maps originally used to access these individual words with new ones which performed the requisite tests to determine whether the words were in the static part or the differential part, redirecting the store accesses accordingly.

In order that the translation table in the differential file should not grow too large the database was arbitrarily broken up into cells of several words. When a word was modified it was written to the differential part and its companion words in the cell were copied to their corresponding sites. Due to the storage scheme used by *Rbase*, if one word is written then it is very likely that the following word will be the next one written. Thus it was very often the case that if one word in a cell were modified then the copy of the following one would immediately be overwritten. As this was not always the case, however, the copying could not be dispensed with.

4.2 Tests performed

Tests were carried out using the modified system described above using a database modelled on that required in the administration of the Computer Science 1 course. There were 250 student records, each of which contained

- Surname
- Prenames
- Tutorial group
- Faculty
- Course (either CS1a or CS1b or a half course, coded as '*')
- Marks for five practical exercises
- EMAS user number (in the range ECXU00 to ECZU99 — these were coded as X00 to Z99 as the EC and U parts were constant)

As *Rbase* imposes a limit of five on the number of columns a tuple may have, the practical marks were grouped together into one sub-tuple, as were faculty and course. User number was stored as the tuple value. The tuples were arranged as follows:

```
[Surname Prenames Tutorial [Faculty Course]
                           [m1 m2 m3 m4 m5]], Uno
```

Before each test run the static part of the database was preloaded with two-thirds of the student records. Each test run then consisted of the loading of the remainder followed by the answering of some queries typical of the type for which such a database would be used. More precisely, each test run consisted of the following sequence of tests:

1. Load the remaining one-third of the student records (83) into the database.

2. Show all students in any of Malcolm Atkinson's tutorial groups. This query is formulated as:

   ```
   getall [?? ?? ?[SUBSTRING mpa] ?? ??], ??
   ```

   This resulted in 18 student records being retrieved.

3. Show all students in any tutorial groups taken by either Paul MacLellan or Paul Cockshott whose surname has initial letter in the range 'A' to 'G':

   ```
   getall [?[< H] ??
             ?[ANY ?[SUBSTRING pmm]
                   ?[SUBSTRING pc]] ?? ??],
           ??
   ```

   10 records were retrieved.

4. Which students in Alec Wight's first tutorial group are in the Science Faculty:

   ```
   getall [?? ?? aswl [S ??] ??], ??
   ```

   9 records were retrieved.

5. Which of Kathy Humphry's tutees have user numbers in the range ECXU00 - ECXU99:

   ```
   getall [?? ?? ?[SUBSTRING kh] ?? ??],
           ?[< Y]
   ```

   11 records were retrieved.

6. Which of Frank Stacey's tutees are doing a half course, and not either of the full courses:

   ```
   getall [?? ?? ?[SUBSTRING fs] [?? *] ??],
           ??
   ```

   4 records were retrieved.

These test sequences were performed on the following four database configurations:

Test *A*	No differential file, all updates being made to the main file, but via the address translation maps. This represents the base case with the minimum overheads being introduced, and is therefore the standard against which the remaining cases should be compared.
Test *B*	Updates being made to the differential file.
Test *C*	Updates being made to the differential file, the translation mechanism modified to

remember the last address translated. This addition was intended to exploit the known tendency of *Rbase* to access store locations near to those accessed immediately previously, as explained above. These might lie in the same cell when the address translation would only need to be performed once.

Test *D* Bitmap filters disabled. All address translations are checked in the differential file including those which would otherwise be known definitely not to reside there.

4.3 The results

The results given in tables 4-1 and 4-2 were obtained with three independent bitmaps, each of 16384 bits, 16 byte (4 word) cells for the differential file, and the translation table arranged as a hash table with a linear scan for an empty cell if the target one is already occupied.[1]

Table 4-1: CPU time (seconds) for *Rbase* tests

Part	Test *A*	Test *B*	Test *C*	Test *D*
1	18.45	46.25	40.82	66.63
2	2.33	4.54	5.61	7.58
3	2.05	3.75	3.89	6.21
4	0.50	0.63	1.57	1.00
5	1.56	3.07	2.80	5.76
6	1.82	4.05	3.31	7.83
Totals	26.61	62.29	58.00	95.01

Table 4-2: Page faults (number) for *Rbase* tests

Part	Test *A*	Test *B*	Test *C*	Test *D*
1	1231	4501	4237	8238
2	219	585	622	865
3	134	575	421	700
4	9	40	33	53
5	201	547	565	681
6	129	555	530	806
Totals	1923	6803	6408	11343

It will be seen that there is a substantial difference, both in CPU time used and in page faults, between the test run without the differential file and those with it. CPU time has more than doubled for tests *B* and *C* over *A*, and that of *D* has increased almost fourfold. Similarly, the number of page faults has increased by more than threefold for tests *B* and *C*, and almost sixfold for *D*.

[1]Measurements made on the Computer Science Department's *VAX-11/780* running *VMS* version 1. Tests were run late at night when the machine was lightly loaded.

The overall elapsed time for each set of tests was also obtained. As there were other users on the machine at the time, however, these figures should not be taken as precise measurements of comparative performance, as is the case for CPU time and page faults which are reasonably independent of other processes, but rather as approximate guides. These figures are given in table 4-3.

Table 4-3: Overall elapsed times for *Rbase* tests

Test	Elapsed times
A	0 mins 32 secs
B	1 min 28 secs
C	1 min 27 secs
D	9 mins 24 secs

Substantial differences will again be seen in these results. The elapsed times for tests *B* and *C* are almost three times that of *A*, while the elapsed time for *D* is almost nineteen times that of *A*. Similar results were obtained subjectively when the system was used interactively. The performance for configuration *A* (no differential file) was quite acceptable, with little real subjective difference being observed from the performance of the original, unmodified system. The performance for configurations *B* and *C* (both with a differential file, *B* without and *C* with the address translation memory) was noticeably poorer, although still usable. There was no subjective difference between these two. The performance for configuration *D* was unacceptably bad, with quite long pauses between the production of successive records. The perceived performance difference between configurations *B* and *C* on the one hand and *D* on the other, together with the differences in measured performance, serve to underline the effectiveness of the bitmap filter in eliminating unnecessary translation table lookups, thereby dramatically reducing the system overheads. It was concluded that such a filter is is essential to the efficient working of a differential file system.

There is no real difference in the measured performance of configurations *B* and *C* for tests 2 to 6 (those involving reading but not writing of the database), but for test 1 (writing to the database) configuration *C* shows a small reduction in both CPU time and page faults over *B*. This reflects the fact that the system, on reading, may only access two or three successive words, whereas on writing, larger areas of consecutive words are accessed. The gains resulting from remembering the last address translation will, therefore, be manifest mainly in writing operations.

4.4 Copying overheads

The difference in measured performance between configuration *A* and configurations *B* and *C* is quite large, due to the overheads involved in maintaining the differential file. If the cell size were increased, copying would account for a greater proportion of the overheads, but there would be a corresponding decrease in the sizes of the bitmaps and translation table required, due to there being fewer cells and therefore fewer different addresses to translate. Furthermore, the copying only occurs when the database is being written to, and not when it is being read. There would, therefore, be an application-dependent point where the overheads were minimised, any decrease in the overheads of the translation table being more than compensated for by increased overheads of copying, and *vice versa*.

It would be desirable if it were somehow possible to decrease the sizes of the bitmaps and the translation table by increasing the cell size, with the decrease in CPU time and page faults which would result, without at the same time increasing the overheads caused by copying the unchanged words of a cell when it was modified for the first time. At first sight this appears an impossible dream. Recall, however, that disc controllers and drives transfer data between main store and backing store in contiguous blocks, typically of 512 bytes. Thus, if, when we modify part of the database for the first time, we alter the in-store copy *in situ* and then arrange that the entire surrounding disc block is written back to a different site than the one from which it was read, then the copying of the surrounding words of the disc block will be done by the disc hardware for us "on the fly". In any case, the higher levels of the database system would tend to group "related" objects together in the same block or frame, and so to preserve the performance gains which this brings it is necessary for the lowest level to preserve the physical arrangement of the database.

Such an increase in cell size, from 16 bytes in the case of the configurations measured above to 512 bytes (a 32-fold increase), would result in a corresponding decrease in the sizes of the translation table and the bitmap, although not to such a great extent. This is because the proportions of modified and unmodified data in a cell would change, with larger cells having a greater proportion of unmodified data than smaller cells. The actual decrease would depend on the pattern of use of disc blocks by the database system. For a system where all the information on a record was localised in one or two blocks the decrease would be comparable in size to the increase in cell size, while for a system, such as *Rbase*, where the information on a record is scattered throughout several blocks the decrease would be less marked.

The techniques used in the *Shrines* system (chapter 7) having not yet been developed, the lack of a clearly defined database system to operating system interface would have made it very difficult to implement differential files based on disc blocks, as described above. In view of this, the subjective and objective performance of the existing system, and the fact that the data model would probably not have lent itself to CAD databases, it was decided to leave the *Rbase* experiment at this point to investigate further differential files based on disc blocks using a purpose-written system.

Chapter Five
A Disc Block Based Implementation

5.1 Overview

This chapter describes an implementation of a virtual file system based on disc blocks. We have seen that there are performance advantages to be obtained by making the block the unit of differentiation. This is due, in part, to the fact that the copying required when a cell is first modified may be eliminated, with the result that the CPU overheads involved in writing to the database may be reduced. Furthermore, it is common practice for a database system to group related items (tuples, records) together, and that if we are to preserve the advantages which this brings then we must do likewise. For example, geometric modelling is not practicable if each object must be fetched individually from disc [Eastman 80]. A partial implementation of a *frame memory* system [March 81] showed that a block-based system would pose no problems for higher layers of software.

Rather than tie an implementation to a specific database system, which although ideal for some applications would be unsuitable for others, it was decided to build a general *virtual file* system which could then be used by different database systems as their transaction and backing store manager, or by applications programs as a substitute for direct-access files.

From the point of view of the user program, the virtual file appears similar to a direct-access file. The user is able to create, open and close it, and read and write blocks from it in any order. In addition to this it is possible to create a differential file layer on top of any virtual file to an arbitrary depth (the only constraints being imposed by the operating system as to the number of physical files allowed to be open at any one time) and to merge these differential files with their fixed parts or to delete them. These functions may be further packaged as a set of procedures to open, suspend, resume, commit or abandon experiments or transactions, and to read and write blocks from them. The user is thus relieved of all requirements to handle the various "real" files directly, but can operate on the virtual files as though they were "ordinary" direct-access files with the added flexibility that experiments provide. A description of the interfaces is given in appendix B.

An implementation, such as this one, based on the philosophy of virtual direct-access files, while possible under almost any operating system, is of most relevance to machines without virtual memory support, such as the Computer Science Department's *advanced personal* *machine* (in its current form) [Dewar 83]; in this case the host filing system would not be local, but would be located on a file-server somewhere on a local area network. Virtual memory support allows a number of additional techniques to be employed to enhance performance and user-convenience; these will be discussed in chapter 7.

The system is written entirely in *Imp77* [Robertson 80], and has been run successfully on a Perkin-Elmer *3220* [P-E 79] under *Mouses* [Culloch 79] and a *VAX-11/780* under *VMS* [DEC 77a, DEC 77b, DEC 78] with no code changes required, apart from a special purpose interface to the host filing system.

5.2 System architecture

The architecture of the system can best be described with the aid of the diagram in figure 5-1 on page 51. This diagram shows the modules from which it is built and the way in which the modules interact with each other; any two modules interact if and only if they have a common boundary in the figure.

At the lowest level, the *file system interface* provides a standard interface for all higher levels, not only for reading and writing, but also for creating, deleting and extending real files.

At the next level is the *disc block cache*, which provides a resident disc block buffer, the size of which is a compile time parameter (although for test and performance measurement purposes the facility was added to selectively disable sections, thereby effectively reducing its size). This holds both blocks of translation data and blocks of user data, with no distinction being made between them, blocks being replaced on a least recently used basis. A previous version of the system had two separate caches, one for user blocks and one for translation blocks. It was found, however, that this could result in thrashing in one cache while there was spare space in the other, with a consequent increase in disc traffic, unless the relative sizes were finely adjusted. It was decided to merge the caches and to allow a natural balance dynamically to establish itself, rather than statically to tune the system for optimality under one set of conditions, only to produce distinctly non-optimal performance under another set of conditions.[1]

At the topmost level the user program interfaces to two independent parts of the system. The *Virtual I/O* section manages the blocks which go to make up a virtual file. When a block is accessed the virtual I/O manager first checks to see whether it is currently cached or not. If it is, no further work need be done. If it is not, then the *translation* section is invoked to bring it into the cache. The *virtual file manager* is responsible for managing the individual real files that go to make up a virtual file. To achieve this it requires access to the cache in order that modified blocks may be flushed when a file is closed, and directly to the filing

[1]The cache algorithms could benefit from hardware or microcode assistance. See appendix E (page 144).

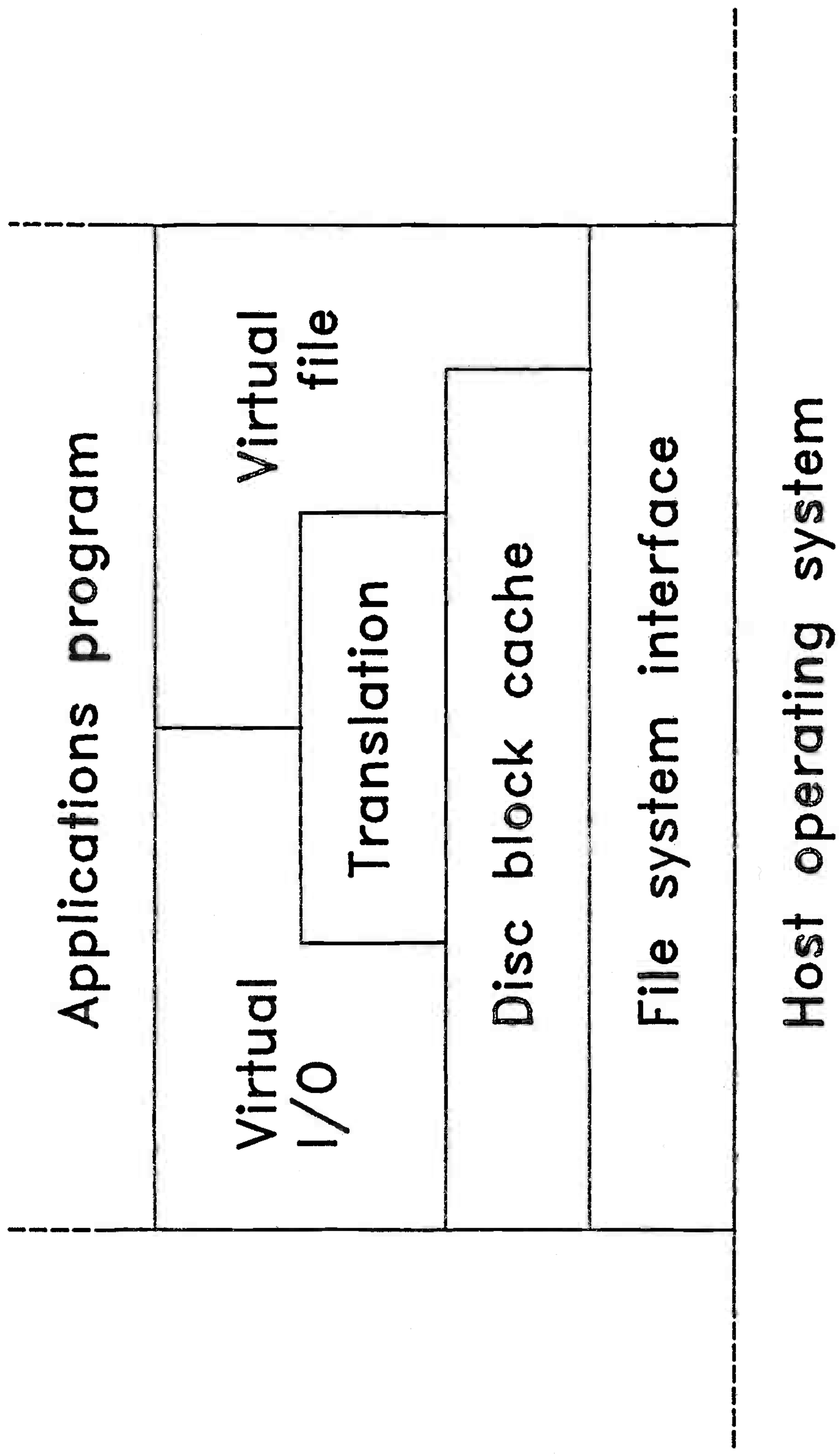


Figure 5–1: System architecture – an overview

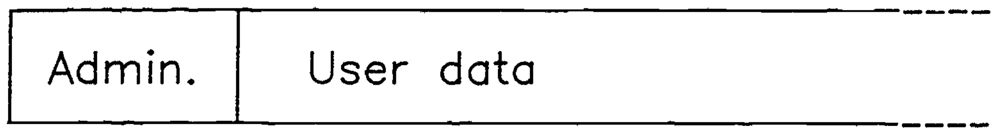
system (via the standard interface) to manipulate the administrative blocks and to create and delete files and modify their attributes.

In addition, the virtual file and virtual I/O sections may invoke each other from time to time: the I/O manager invokes the file manager to allocate space in a file, extending it if necessary; the file manager invokes the I/O manager when merging a differential file with its static part to ensure that all the necessary translation table entries are set up correctly, as the topmost layer of the static part will, in general, itself be a differential file (namely the parent experiment).

5.3 File layouts

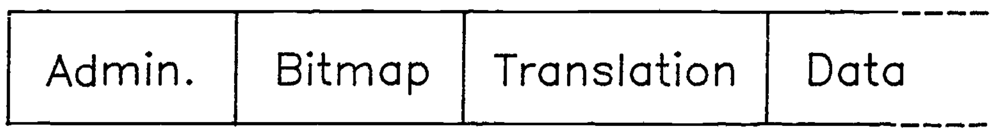
Each layer of a virtual file occupies one real file. At the lowest level all virtual blocks are present as real blocks and no translation is required other than to add a constant offset to allow for the administrative block at the start. The layout of a file at this level is given in figure 5-2.

Figure 5-2: File layout - lowest level



For all higher layers a virtual block may or may not be present. If it is, then a translation is required to map it onto its corresponding real block, as described previously. Thus, in addition to the administrative block and the user data, a filter bit-table and a translation table are also required. The layout of the file then becomes as shown in figure 5-3.

Figure 5-3: File layout - higher levels



5.3.1 The filter bit-table

The filter bit-table is one block in size. Although block size was made a compile time parameter to allow for use under different operating systems, both machines on which the system has been tried have a block size of 512 8-bit bytes giving a filter bit-table size of

4096 bits. Only a single hashing function was used in conjunction with the filter bit-table, namely masking off the low order bits of the virtual block number with 4095. With this hashing function and virtual files of size no greater than 4096 blocks, the fact that a bit is **set** is a positive indication that the virtual block **is** to be found at the current level, while the fact that it is **unset** is a positive indication that the block **is not** to be found at the current level (in other words, the filter acts as a "normal" bitmap). For larger virtual files the filter bit-table wraps round, and the fact that a bit is set indicates only that the virtual block **may** be found at the current level.

The system has been tested with virtual files substantially larger in size than 4096 blocks as well as with smaller ones, and works successfully in both cases. It was thought unlikely, at least in the environment in which the prototype system would be operating, that many files would exceed 4096 blocks in size, and hence that no more complex arrangement would be required.

5.3.2 The translation table

As with the Rbase implementation, the translation table is arranged as a hash table. Unlike that implementation, however, where the address was rehashed if it collided with another entry in the table, in this implementation overflow chains are used to deal with collisions. This has the advantage that the size of the differential file is not constrained by the size of the hash table, as was the case with the Rbase implementation, but can grow arbitrarily large. When an entry must be put onto an overflow chain the next free slot in the current overflow block is used. If this is full then the next free block in the file is claimed, in the same way that a new data block would be claimed for user data. Thus, translation blocks and user data are freely intermingled throughout the remainder of the file. Following the overflow chains will not impose much of an overhead on the system since, being frequently accessed, the blocks of the translation table will tend to remain resident in the cache. Again, the hashing function was kept simple, being merely the virtual block number *modulo* the size of the hash table. The size of the hash table may be set when the differential file layer is created, a default value, chosen at compile time, being used if this is not done.

The amount of translation space required by this scheme should be no greater than the amount required by the scheme used in the Rbase implementation. Indeed, the total space dedicated to translation tables in the latter scheme may be substantially larger due to the fact that the hash table cannot be allowed to become more than about half full for reasons of access efficiency. In any case, with the latter scheme, the hash table must be sufficiently large to cope with the largest differential file likely to be encountered, and will therefore tend to be substantially larger than that actually required to cope with files of a more "normal" size.

5.3.3 Administration

The administrative area occupies a single block, the space being divided equally between information required to manage the physical file being used at the current level and information required to manage the virtual file as perceived from the current level. The latter area would normally contain information required by the database system, such as pointers to dictionaries and free lists, while the former area will include such information as the size of the physical file, the location of the next unused block, and, if the file is a differential file, the name of the fixed part and the location of the current translation overflow block and the next free slot therein. In addition, the size of the translation hash-table must be stored, since this may be different for different parts of a virtual file.

All the filter bit-tables and administrative blocks are resident in main store while a virtual file is open, the space being allocated from the heap and restored to it when the file is closed. The space for the cache is also allocated from the heap when the first virtual file is opened, but remains allocated until the program terminates.

The results of a series of performance tests on this system will be presented in the next chapter.

Chapter Six
The Performance of the Block-based System

6.1 Test philosophy

Having constructed the virtual file manager based on disc blocks described in the previous chapter it was then necessary to evaluate its performance. As indicated in chapter 4, the performance of the *Rbase* based system was rather poor, unacceptably so when the filter was disabled. There would be little point in using a virtual file based system if its performance were no better than that of file copying, and although, subjectively, the performance of the virtual file system appeared to be quite satisfactory (indeed there was little, if any, difference in performance perceptible as layers of differential files were added) it was felt that a more objective measure of system performance should be obtained.

As the basic philosophy of the system was that it should be used as a file manager on which applications could be built, it could not be tested using "typical data" as was the case with *Rbase*. Instead, each application would have its own pattern of use, possibly radically different from all others. It was decided that, rather than attempting to guess at possible use patterns, a pseudo-random number generator should be used to control the tests (a linear congruence generator, supplied by Gordon Brebner). This was used to choose a random block to be read from or written to the virtual file.

This is a severe test, as particular blocks tend to be accessed comparatively infrequently thereby diminishing any benefits which might otherwise accrue from the cache with a kinder access pattern. With a real application we would expect that each block would be accessed several times; this does not happen with the random access pattern used here. Furthermore, with the increased locality of access to be expected from a real application we would expect that the required translation blocks would be found in the cache more frequently, by virtue of having been brought in there when a neighbouring block was accessed; again, this does not happen in our tests. Thus, although it could be argued that the results do not correspond to a "typical" pattern of use, it is the case that the overheads apparent with such a more "typical" pattern would tend to be less than with the random number generator, and that if the performance were found satisfactory with the random number generator then it would almost certainly be satisfactory for real applications.

In addition, although the blocks were being transferred to and from disc, no processing was carried out on the data contained therein. This would not normally be the case for a real application, however, and so the processing performed by the virtual file system accounts for a disproportionately large fraction of the total processing as compared to a real application.

All the tests were performed on the Data Curator Project's *Perkin-Elmer 3220* running *Mouses* V9.2 SS 3.2/1.4 as a single-user system, with no extra delays and overheads caused by swapping or competition for CPU and disc resources. A complication arises in that the machine has store boards from two different manufacturers with different speeds of memory chips. In the absence of swapping, since there were no other users to induce it, a program remains resident in store where it has been loaded. Hence, although different test runs are not comparable in respect of timing measurements, since the supervisor may have loaded them into different parts of memory with a corresponding difference in execution speed, nevertheless individual test runs are self-consistent, since the pattern of memory usage remains fixed throughout.

The basic performance test was also run on the Computer Science Department's *VAX-11/780* under *VMS* V2.4 on a lightly loaded system. It was found, however, that although the non-timing measurements were substantially identical to those of the *3220*, there was so much variance in the timing measurements that the results supported only substantially weaker conclusions. In view of this, and the difficulty in finding times when the machine was sufficiently lightly loaded that the results were at all meaningful, it was decided to perform the remaining performance measurements on the *3220* where the conditions were much more conducive to obtaining accurate measurements

6.2 Measurements obtained

The following measurements were taken during each test:

- **CPU time** (in *milliseconds*). This was obtained as the difference of the test process's CPU times before and after each test. On the *3220* under *Mouses* this figure is given with a resolution of one millisecond (although the operating system holds it to an order of magnitude higher resolution), the actual value being determined by resetting the interval timer when a user process is scheduled, starting it immediately before a user process is put on CPU, stopping it immediately any interrupt occurs, and reading the elapsed interval from the internal register and charging the user process whenever another process is scheduled [Culloch 82]. On *VAX* the resolution is ten milliseconds, the interval between hardware clock interrupts, with the entire period being credited to the currently-executing user process (if any) by the EXE$HWCLKINT interrupt service routine, and hence the value can not be regarded as more than an approximate indication of the actual CPU time usage of a process [DEC 81].

- **Elapsed time** (in *seconds* with a resolution in seconds). This was obtained on both *VAX* and the *3220* from the time-of-day clock.

- **Blocks read.** This is the total number of disc reads performed, as tallied by the file system interface procedures.

- **Blocks written.** This is the total number of disc writes performed, again tallied by the file system interface.

- **Cache hits.** As described in chapter 5, the virtual I/O manager first searches the disc cache in case a block is already resident, in which case no further work need be done. The translation manager also searches the cache for parts of its translation tables. This figure represents the total number of times disc blocks were found to be resident, irrespective of whether they were user or system data.

- **Cache misses.** As with "cache hits", this figure represents the total number of times the required disc blocks were not found. Note that a cache miss does not necessarily imply a corresponding disc transfer, since the virtual I/O manager searches the cache before invoking the translation manager. Thus, if a virtual block is not resident it will cause **two** cache misses, one for the search for the virtual block and one for the fetch of the corresponding physical block after address translation has been performed, in addition to any misses caused by translation blocks being non-resident.

- **False positives.** This figure represents the number of times the bitmap filter of the differential file wrongly indicated that a block might be present. As explained in chapter 5, for files of size no greater than 4096 blocks this provides a **definite** indication of presence or absence of a block. As this condition held for all tests except those designed to measure performance with large files or those where the filter was disabled, there were no false positives in most tests, and the figure has only been given where it was non-zero.

All the results shown in the tables are means obtained from 50 replications,[1] with the corresponding standard deviations given in "( )". Except for tests specifically designed to alter them, the default conditions for the tests were: cache size 24 blocks; translation hash-table size 4 blocks; bitmap filter enabled; cache discipline "least recently used"; random block numbers in the range 1..4096.

6.3 Basic performance test

The first test performed was a basic performance test to establish whether the overheads incurred by using differential files were acceptable. This test was in four parts, as follows:

Test *A*. Create a new file at the lowest level (one where there would be no need of translation). Write 5000 randomly chosen blocks and then read them back again. Write 1000 more blocks and then read them back again.

Test *B*. Create a new file, write 5000 blocks and then read them back again. Create a differential file layer, write 1000 blocks and read them back again.

[1]Except for one test where a disc hardware fault resulted in the loss of one replication.

Test *C*. Create a new file and write 5000 blocks to it. Create a differential file layer and write 1000 blocks to it. Read the original 5000 blocks and then the remaining 1000 blocks. (This sequence of operations was used as the basis of all subsequent tests.)

Test *M*. Merge the 1000 blocks from the differential file into its static part.

Updating 20% of the database would be an infrequent occurrence in "normal" usage; however since we are interested in comparing the effects on the access times to the differential part this number was made artificially high.

Note that the blocks are not necessarily all distinct; indeed, it would be *impossible* to choose 5000 distinct blocks from the range 1..4096. If blocks are chosen at random there may be some which have not been chosen at all, some which have been chosen once, some twice etc. The random number generator was used to choose 5000 blocks in the range under consideration and the number of times each was chosen was tallied. The resultant empirical distribution is given in table 6-1, which lists the expected numbers of blocks, of a total of 5000 choices, which would be chosen given numbers of times (averaged over 1000 replications).

Also given in table 6-1 are the expected numbers of blocks which would be chosen assuming a Uniform distribution over the range of blocks. It follows from [Feller 68] chapters IV and IX that if *r* blocks are chosen from the range 1..*n*, then the expected number of choices which have been chosen *k* times is

$$\lambda = E_k(n, r) = n \exp(-r/n) (r/n)^k / k! \qquad (6.1)$$

This result is also derived in [von Mises 64] chapter 4 section 9 in the context of the asymptotic behaviour of occupancy problems.

Table 6-1: Duplication rates choosing 5000 numbers from 1..4096

Chosen	Empirical	Theory
1	1580.0	1475.1
2	992.9	900.3
3	344.9	336.3
4	80.4	118.8
5	13.4	27.3
6	1.6	5.6
7	v. few	1.0
$\geqslant$ 8	v. few	v. few

Thus, for example, if we choose 5000 blocks from the range 1..4096 using the pseudo-random number generator we would expect that about 345 of them would have been chosen 3 times.

It will be seen that the results obtained using the random number generator show more numbers being chosen once or twice than would be expected if the numbers were being chosen from a Uniform distribution, with a corresponding shortening of the tail. This is explained by the cyclic properties of the type of generator used which will tend to introduce a "memory" factor into the choice, so that once a number has been chosen it will tend not to be chosen again for some time, in contradistinction to the theoretical case where the choices are assumed to be made independently. This effect will tend to make the tests more severe than would be the case if a theoretically Uniform random number generator were used, since the benefits of the cache will be reduced even further. Furthermore, we should use the empirical distribution in assessing the results obtained, since this has more relevance to the actual conditions of the test runs, rather than the theoretical distribution of equation (6.1); all the tables of expectation given in this chapter show the empirical distribution rather than the theoretical distribution of equation (6.1).

Similarly, choosing 1000 blocks gives table 6-2.

Table 6-2: Duplication rates choosing 1000 numbers from 1..4096

Chosen	Empirical	Theory
1	805.1	783.4
2	88.8	95.6
3	5.2	7.8
4	0.4	0.5
$\geq$ 5	v. few	v. few

The fact that a block may have been chosen more than once is of no relevance to the *write* parts of the tests, since the block was claimed anew each time rather than read from disc. For the *read* parts of tests *A* and *B* some of the blocks which have been chosen more than once may still be resident in the cache on their second selection. This will not be a frequent occurrence, however, since the cache is small compared to the number of blocks being read. The entry for *cache hits* in column *Test A* of table 6-4 (page 60) indicates that this was the case for only about 31 of the 6000 blocks being read back, or about 0.5%.

For test *C* it will also be the case that some of the blocks chosen in the second 1000 will also have been chosen (possibly more than once) in the first 5000. In this case the block will appear to be in the differential part rather than the static part during the re-read of the first 5000 blocks. Thus, during this re-read, some of the blocks will be read from the static part (no translation necessary), with the rest being read from the differential part. In order to determine how frequently this would occur 5000 blocks were chosen at random, followed by a further 1000. The occurrences of blocks in the first 5000 which also appeared in the second 1000 were tallied, giving table 6-3 (again averaged over 1000 replications).

Table 6-3: Expected duplication rate of 1000 blocks in 5000

Duplicate	Expect
No	4024.6
Yes	975.4

From this table it will be seen that about 975 random choices from the first 5000 will also appear in the second 1000, which is a density of 20%, a not inconsiderable fraction.

6.3.1 Results of the basic performance tests — *3220*

The results obtained from the basic performance tests as run on the *3220* are shown in table 6-4.

Table 6-4: Basic performance test — *3220*

	Test *A*	Test *B*	Test *C*	Test *M*
CPU time (milliseconds)	44428.6 (5.796)	47738.7 (65.68)	52281.0 (115.3)	6487.9 (56.84)
Elapsed time (seconds)	332.90 (0.974)	340.60 (3.580)	384.58 (3.044)	61.12 (0.872)
Blocks read	5970.4 (2.948)	6555.9 (25.03)	8477.2 (65.40)	1293.8 (20.96)
Blocks written	6002.0 (0.000)	6178.0 (7.958)	6177.3 (7.475)	901.9 (7.407)
Cache hits	30.6 (2.948)	6394.0 (95.52)	7947.3 (139.9)	507.2 (11.80)
Cache misses	5969.4 (2.948)	7554.7 (25.02)	14476.6 (65.36)	391.9 (15.29)

Total elapsed time for test: 15h 51m 17s

Test *A* provides a baseline measure of the performance of the virtual file system with the test program, and represents the case where a virtual file comprises only one real file with only minimal translation being required.

In test *B* the second 1000 blocks come from the differential file. The increase in overheads can be seen in the increase of 7% in CPU time and of 10% in total disc traffic. The extra CPU time has been required to perform the processing required by the translation. The increase in disc traffic is caused by the requirement to read non-resident translation blocks, possibly flushing a different translation block to make room. The effect of the cache can be clearly seen in the cache hits figure which has increased over 200 fold, while the cache misses have increased by only about 10%. Total elapsed time has increased by 3%. The access pattern of this test would be expected to be highly unusual in practice, however.

Test *C* gives a more realistic access pattern since, as we saw above, 20% of the blocks re-read from the first 5000 will come from the differential file, giving an intermingling of reads from the base file and differential files. Tests *B* and *C* have in common the writing of the original 5000 blocks and the reading and writing of the second 1000 blocks. The difference between them is that in test *B* the first 5000 blocks are read back from the base file, while in test *C* they are read back through the differential file with some of them coming from the differential file and the remainder from the base file. Hence the difference between tests *B* and *C* represents the difference between the base case with minimal translation, most nearly resembling a "normal" file, and the general virtual file case. Test *C* most nearly resembles what might be expected as a "typical" access pattern, namely intermingling of reads from the fixed part with operations on the differential part, except that the cache is rendered rather less effective than would otherwise be the case since particular blocks are accessed so infrequently.

Again, CPU time has increased, this time by 9%, the difference being caused by the translation requirements of those blocks, expected to be around 975 in number, which must be read from the differential part rather than the fixed part. We would expect that the translation blocks would be flushed out of the cache more frequently than was the case with the second 1000 blocks of test B, since in that case all 1000 blocks required translation and so translation blocks would be frequently used and would tend to remain resident in the cache, while here only 20% require translation, the remaining 80% flushing out translation blocks along with "user data" to make room in the cache. This effect shows up in the blocks read measurement which has increased by about 30%. Another factor, which is by no means negligible as will be seen later (section 6.10.1 on page 72), is the increase in head seek time occasioned by the increase in total size of the files being accessed during the re-read of the first 5000 blocks. The *MSM80* drives attached to the *3220* hold 160 blocks/cylinder, and so the number of cylinders required has increased from the 25 required to hold the base file by the 7 or so required to hold the differential file, or by almost 30%. With more cylinders holding the file, the chances of the heads being off-cylinder has increased, as has the average seek distance, resulting in more frequent and longer seeks.[1] As a consequence of all these factors, total elapsed time has increased by 13%.

No direct comparison exists between test *M* and the other three tests, since they have no part in common. A comparison with the *COPY* command provided by the subsystem for copying files is revealing, however. This command is optimised to reduce head movement by remaining on-cylinder to transfer as many blocks as possible, rather than copying block by block with a resultant increase in head movement. The *COPY* command takes about 38 seconds to copy a 1000 block file, as against the 61 seconds taken to merge the second 1000 blocks with the first 5000, over 60% more. This reaffirms the contribution which head movement and seek time make to the overall elapsed time. It should be remembered, however, that in a real application the access pattern would probably be less haphazard than the random one of the experiment, and that seek times would be correspondingly less.

Furthermore, as described in chapter 3, the cost of committing an experiment using differential files is the cost of a merge, although there is no reason why the user cannot proceed immediately, leaving the merge to take place as a background task; in contrast, the cost of starting a transaction is the cost of copying the database using file copying, with the user being forced to wait until this is carried out. In this case the comparison is between

[1]It was known that the algorithm used by the file system would place the base file and the differential file adjacently. Clearly, if this had not been the case the effect due to head-seek time would have been even greater.

merging 1000 blocks, at a cost of 61 seconds elapsed time, as against copying at least 4000 blocks (the static part which, since we are choosing blocks from the range 1..4096 must be at least this size), at a cost of 160 seconds elapsed time.[1] Thus, in this case, committing an experiment using differential files is less than half of the cost of opening just one experiment using file copying, although as the user may proceed immediately in the former case but not in the latter the apparent discrepancy in cost is even greater.

6.3.2 Results of the basic performance tests — *VAX*

The results from the test performed on *VAX* show the same pattern as those obtained from the *3220* (table 6-5).

Table 6-5: Basic performance test — *VAX*

	Test *A*	Test *B*	Test *C*	Test *M*
CPU time (milliseconds)	46245.2 (1144)	49946.4 (1165)	56307.8 (1394)	7297.8 (279.4)
Elapsed time (seconds)	417.52 (20.32)	428.60 (25.64)	482.52 (27.54)	78.74 (4.759)
Blocks read	5970.5 (3.253)	6556.2 (24.93)	8468.0 (59.63)	1293.2 (19.71)
Blocks written	6002.0 (0.000)	6175.9 (7.712)	6176.1 (8.638)	901.4 (7.421)
Cache hits	30.5 (3.253)	6386.3 (85.82)	7941.5 (124.3)	506.9 (11.40)
Cache misses	5969.5 (3.253)	7555.1 (24.88)	14467.2 (59.73)	391.8 (14.32)

Total elapsed time for test: 19h 32m 49s

The results of the non-timing measurements (blocks read, blocks written, cache hits, cache misses) are virtually identical with those obtained on the *3220*, which is as we would expect given that the programs are identical. The timing measurements show substantially higher standard deviation, however, and the conclusions we are able to draw are correspondingly weaker. The elapsed times for tests *A* and *B* are within a standard deviation of each other, for example, and hence there is no evidence that there is any difference in elapsed time between tests *A* and *B*. That the CPU time results from the *3220* run are consistently lower than those of the *VAX* run, despite the fact that the former is a slower processor than the latter, can be accounted for partly by the superior code quality produced by the *3220 Imp77* compiler as compared to the *VAX* one, and partly by the contamination of the *VAX* results with system overheads. Elapsed time for *VAX* is also consistently higher than for the *3220* by a

[1]Because of increased head movement this is not just four times the cost for 1000 blocks.

factor of about one quarter. This can be accounted for partly by the presence of other users on *VAX*, although as the test was run late at night this effect should not be large, and partly by the increased disc access times of the *RK07* drive used for the *VAX* test as compared with the *MSM80* drive used for the *3220* test.

6.4 Translation tables

At this point we pause to consider the structure of the translation tables after having 1000 random blocks written to the differential file. Of interest are the lengths of the overflow chains and the number of blocks which each overflow chain resides on. The greater the former, the more processing required to follow each chain. The greater the latter, the more disc transfers likely. Table 6-6 gives the expected number of virtual blocks whose translation will require that a chain be followed down the specified distance (including the original hash-table entry).

Table 6-6: Overflow chain length (entries)

Distance	Expect
1	168.0
2	166.0
3	158.2
4	139.4
5	109.0
6	75.5
7	44.2
8	21.7
9	11.4
10	5.2
11	2.6
12	1.9
13	0.6
14	0.1
≥ 15	v. few

This table, giving means for 10 replications, was obtained from a printout of the translation chains in the differential file after writing 1000 blocks to it. About 139 blocks will require translation chains to be followed to depth 4, for example. It is interesting to note that 90% of blocks will require chains to be followed to depth 6 or less. Hence, even with such a large number of updates as 1000, the processing required to follow translation overflow chains will not be excessive.

Similarly, table 6-7 shows the expected numbers of virtual blocks which will have translation chains residing on the corresponding number of overflow blocks (again, including the original hash-table entry). Here, 87% of virtual blocks will require 4 or fewer overflow blocks to be accessed. As the blocks corresponding to the start of each chain will be accessed more frequently than those to the end, since they are accessed both for short chains and for long chains, whereas those at the ends are accessed only for comparatively long chains, they are likely to remain resident in the cache. Thus, the amount of disc traffic engendered by the translation scheme should not be excessive.

Table 6-7: Overflow chain length (blocks)

Blocks	Expect
1	168.0
2	192.3
3	175.3
4	148.0
5	106.0
6	64.4
7	30.8
8	11.7
9	4.7
10	1.7
11	0.6
12	0.2
13	0.1
> 14	v. few

6.5 The effect of the bitmap filter

Recall, now, the substantial contribution made by the bitmap filter to the performance of the *Rbase* implementation (chapter 4), where it was concluded that without the filter the performance would be unacceptably poor. To determine the contribution of the filter to the block-based file system, part *C* of the basic performance test was performed with the filter both enabled and disabled. The results are summarised in table 6-8.

Table 6-8: With and without bitmap filter

	with	without
CPU time	52269.5	60517.6
(milliseconds)	(103.0)	(179.8)
Elapsed time	384.90	481.68
(seconds)	(3.78)	(6.950)
Blocks read	8472.5	11757.8
	(59.85)	(154.9)
Blocks written	6118.0	6178.1
	(7.482)	(8.674)
Cache hits	7947.6	25388.6
	(104.0)	(162.0)
Cache misses	14471.7	17757.4
	(59.85)	(154.9)
False positives	0.0	4026.8
	(0.000)	(23.81)

Total elapsed time for test: 12h 16m 36s

The contribution is most immediately seen in the number of false positives registered, which has increased from zero to 4027, which corresponds excellently with the expected number of blocks from the first 5000 which would not be in the second 1000 (see table 6-3, page 59). The threefold increase in cache hits is caused the translation blocks near the head of the overflow chains being accessed much more frequently. There has also been an increase in the number of cache misses, due to the fact that it is necessary to follow a translation overflow chain to its end in order to determine whether a given virtual block requires translation or not. In consequence of this there has been a 40% increase in disc reads and a 16% increase in CPU time, resulting in an overall increase of 25% in elapsed time. This serves to reaffirm the important contribution which the bitmap filter makes to the performance of the system.

6.6 The effect of varying the cache size

Although subjectively we expect that increasing the cache size will result in a reduction in disc traffic, we must now attempt to quantify the benefits it brings. Table 6-9 summarises the results obtained by repeating test C with the cache size set to 8, 16, 24, 32 and 40 blocks.

Table 6-9: Different cache sizes

	8	16	24	32	40
CPU time (milliseconds)	47381.8 (95.20)	49599.6 (74.37)	52252.4 (95.59)	55068.7 (119.5)	57612.4 (178.4)
Elapsed time (seconds)	474.90 (6.393)	419.80 (5.241)	384.22 (2.787)	368.02 (2.511)	363.32 (4.796)
Blocks read	12478.6 (154.9)	9969.5 (81.81)	8465.8 (59.62)	7680.5 (38.32)	7266.2 (26.06)
Blocks written	6742.6 (13.74)	6364.0 (9.760)	6177.6 (8.038)	6075.2 (5.877)	6038.4 (3.009)
Cache hits	3952.7 (95.00)	6462.7 (108.8)	7937.0 (129.5)	8713.1 (132.8)	9159.9 (148.1)
Cache misses	18478.2 (154.7)	15968.7 (81.84)	14465.2 (59.52)	13679.8 (38.42)	13265.3 (25.91)

Total elapsed time for test: 1d 4h 28m 12s

As expected, increasing the cache size has resulted in an increase in cache hits together with a decrease in cache misses. This has resulted in a decrease in both disc reads and disc writes, and so, although CPU time has increased linearly with cache size, due to the extra processing required to search the cache, overall elapsed time has decreased with increasing cache size. Note that for a larger cache, however, the amount of benefit to accrue by increasing the cache size is less than that for a smaller cache. Increasing the cache size from 8 to 16 blocks, for example, has resulted in a decrease in elapsed time by a factor of 12%, whereas increasing from 24 to 32 blocks has resulted in a decrease of only 4% and the decrease resulting from increasing from 32 to 40 blocks is about 1%, and in any case only just exceeds one standard deviation. This is as we would expect, since with a small cache only the most frequently accessed blocks will remain resident. As the cache size increases, so less frequently accessed blocks will start to remain resident. Eventually some blocks in the cache will be accessed so infrequently that the overheads incurred in keeping them in the cache exceed those which would be incurred in bringing them in.

In a real application we would expect that the access pattern would be such that blocks would be accessed a number of times in a short period and then left untouched for a much longer period, rather than being touched once only, as here. In this case we would increase the cache size so that these data blocks could remain resident during the period of their use. The higher-level software could make use of its knowledge of the access patterns to the database to indicate to the cache manager that it would not be requiring a particular block in the immediate future, with the result that blocks would remain resident only while they were required (see section 6.7, where the effects of this strategy are measured).

There are other reasons for not increasing the cache size, apart from the diminishing returns it brings and the eventual crossover point where cache maintenance overheads exceed disc transfer overheads. Cache blocks are claimed from the heap when the first cache access is made. On the *3220*, with a limited number of segments of limited size and no paging, a large cache reduces the amount of heap space available to the user. Furthermore, a process must be completely resident before it is allowed to run, and a large cache brings the process nearer to the limit of physical memory available to the user (currently 387 Kbytes). On *VAX*, a paged machine, there is no real limit to the size of the cache imposed by the machine or the operating system, though an increase in cache size, while reducing the number of explicit disc transfers may simultaneously result in an increase in implicit transfers due to paging. This is discussed further in chapter 7.

6.7 Cache disciplines

It will often be the case that the applications software will know that a particular block is not going to be required again in the immediate future. If this is the case, then it should be possible to reuse that particular cache slot rather than toss out the least recently used block, perhaps to read it in again almost immediately. Hence one of the interface procedures allows the user modify the system's treatment of a particular block. (Such a facility has been found desirable by other builders of database systems, e.g. [Sherman 76, Brice 77, Stonebraker 81].) The modes selectable are

- ***Toss Immediate***. The block will be considered as a candidate for replacement before any others in the cache. Note, however, that the block is not immediately written out (or forgotten, if it has not been modified); this will only happen when space is required and there are no more eligible slots. In effect the block becomes the "least recently used".

- ***Least Recently Used***. If a slot in the cache is required and there are no more eligible candidates, the the least recently used block will be written out (if it has been modified) and its slot reused. This is the default behaviour of the system.

- ***Lock in cache.*** The block is made resident and will not be a candidate for removal when space is required. This facility was provided for use in critical sections of code, but should clearly be used with care as it could seriously degrade system performance.

Tests were performed to compare the "toss immediate" mode and the "least recently used" mode. The results are given in table 6-10.

Table 6-10: Cache disciplines

	LRU	Toss
CPU time (milliseconds)	63945.3 (85.47)	67050.4 (1240)
Elapsed time (seconds)	403.56 (7.432)	355.00 (8.018)
Blocks read	8476.8 (55.75)	6000.7 (1.491)
Blocks written	6178.1 (8.617)	6028.2 (0.370)
Cache hits	7940.2 (136.8)	22412.2 (147.3)
Cache misses	14457.5 (55.56)	11998.2 (2.828)

Total elapsed time for test: 10h 44m 53s

As expected, the results of the "toss immediate" discipline show an improvement over those of the "least recently used" discipline. Total I/O has decreased by 17%, with the result that the elapsed time has decreased by 12%. CPU time has, increased slightly, due to the extra processing required. Thus, by indicating to the cache manager that a particular block is no longer required and that the space can be reused, higher level software can effect quite a considerable improvement in performance.

6.8 Variable sizes of translation tables

Table 6-11 shows how the size of the translation hash-table affects performance. For this test the size of the table was set to be 4, 8, 12 and 16 blocks. On examining the total elapsed times, we discover the at first sight surprising result that there is virtually no difference in the results for different sizes of table. In fact, the results are so similar in relation to their standard deviations that we are unable to conclude that the observed differences are other than a random effect. The implication is that, for the translation scheme used here, in conjunction with the cache, it makes no difference at all to the performance of the virtual file system with 1000 modified blocks which of the values we use. We must seek an explanation for this behaviour.

Table 6-11: Translation hash-table sizes

	4	8	12	16
CPU time (milliseconds)	64204.0 (93.54)	62859.0 (61.40)	62448.7 (53.33)	62268.4 (53.70)
Elapsed time (seconds)	406.61 (3.999)	404.00 (1.841)	406.88 (3.889)	410.10 (3.327)
Blocks read	8478.9 (66.08)	8247.8 (47.54)	8227.9 (45.40)	8282.4 (41.15)
Blocks written	6177.7 (8.088)	6318.5 (10.73)	6425.4 (11.61)	6507.4 (12.28)
Cache hits	7939.8 (117.7)	4276.3 (78.38)	3008.6 (60.15)	2294.8 (50.05)
Cache misses	14478.3 (66.07)	14246.9 (47.74)	14226.9 (45.28)	14281.5 (40.85)

Total elapsed time for test: 23h 3m 42s

Consider how the translation scheme would be expected to perform as the size of the translation table was altered. For a small table there would be a small number of initial entries, each of which would have a lengthy overflow chain. For a large table there would be a large number of initial entries, each of which would have a short overflow chain. Note that, for a given number of blocks to be translated, there will be the same number of translation entries required, irrespective of the size of the translation table. Whether they are spread thinly over a large table or thickly over a small one, the overflow blocks will still be accessed sufficiently frequently that they will remain in the cache, although each individual overflow block will be accessed less frequently as size of the translation hash-table is increased, due to the shortening of the overflow chains. In particular, the blocks which hold entries near the start of the overflow chains will be accessed proportionally more often as the table size is reduced, as more translations will collide and require chains to be followed. Furthermore, if the density of entries in the base table is too low, blocks in it may be accessed insufficiently frequently to prevent them being flushed out of the cache.

This behaviour is confirmed by the cache hits figures, which decrease as the size of the translation table increases, since as the overflow chains become shorter so fewer overflow blocks will be accessed (each new overflow cell requires a cache lookup since it may not be on the same block as its predecessor). The overflow blocks are still accessed sufficiently frequently, whether with few long chains or many short chains, however, that they remain in cache, with the result that the overheads change little. We would expect, however, if this analysis is correct, that with a smaller number of blocks in the differential part, increasing the size of the table might tend to result in an overall worsening of performance, as infrequently-accesed overflow blocks would be flushed out of the cache; this effect was apparent, though to a small degree, when the test was repeated for a smaller scale of updates (section 6.11.4 on page 76).

6.9 Several layers of differential files

The tests we have considered above have all involved only one layer of differential file. One of the requirements of a system to support experiments is that the tree should be able to grow to any depth and hence the overheads generated by adding successive layers of differential files should not be excessive. To check this, test *C* was performed with 0, 1, 2, and 3 layers of differential files, with 0 corresponding to test *A* (the base case) and 1 corresponding to test *C*. The results are given in table 6-12.

Table 6-12: Multiple levels of differential files

	Test 0	Test 1	Test 2	Test 3
CPU time (milliseconds)	44415.1 (6.243)	52258.3 (115.3)	52644.7 (93.85)	53008.5 (93.66)
Elapsed time (seconds)	333.08 (6.040)	386.38 (5.900)	386.52 (3.358)	387.98 (5.004)
Blocks read	5969.1 (3.274)	8473.3 (64.28)	8470.8 (64.45)	8470.0 (48.63)
Blocks written	6001.0 (0.000)	6179.7 (7.710)	6184.0 (7.907)	6189.4 (8.084)
Cache hits	30.9 (3.274)	7941.1 (136.7)	7937.4 (139.0)	7936.1 (138.7)
Cache misses	5969.1 (3.274)	14472.5 (64.35)	14470.0 (64.60)	14469.1 (49.04)

Total elapsed time for test: 21h 11m 33s

Comparing table 6-12 with 6-4 shows that the results for 0 layers of differential files are identical (apart from random perturbations) with those of test *A*, while the results for 1 layer of differential file are identical with those of test *C*, as expected. Furthermore, apart from a slight increase in CPU time as the number of layers is increased from 1 to 3, the results for the tests with non-zero numbers of layers are identical (again, random perturbations aside). As the filter is being 100% effective, the only additional processing required is that it should be checked at each level, hence the small increase in CPU time. For virtual files larger than 4096 blocks there would be a chance that the bitmap filters would indicate false positives. If files of this size were to be used regularly, however, it would make good sense to increase the sizes of the filters, since the number of blocks required to give a definite yes/no decision as to the presence or absence of a block is very small compared to the total amount of data stored. We conclude that adding extra layers of differential file causes a negligible increase in overheads.

6. 10 Large files

Let us now consider further the effect on performance caused by a virtual file whose bitmap filters are "not big enough".[1] We do this by comparing performance with random blocks in the range 1..4096 and in the range 1..16384, the former giving rise to a definite indication from the bitmap filter with the latter indicating only "maybe present". We must first compare the expected distributions of the frequency of block selection for these two ranges. Table 6-13 shows the expected numbers of blocks out of 5000 which would be chosen from the range 1..16384 one or more times.

Table 6-13: Duplication rates choosing 5000 random numbers from 1..16384

Chosen	Expect
1	4237.0
2	381.5
≥ 3	v. few

This should be compared with the table for the range 1..4096 (table 6-1, page 58). We see immediately that the incidence rate of blocks being chosen several times has, as expected, fallen greatly. Whereas for the smaller files only 32% of blocks would be chosen only once, for the larger files only 8% would be chosen more than once. This is not of great consequence to the tests, however, as 5000 blocks will still be written as new whether or not they are distinct, with neither more nor less effort being required for the minimal translation process. For the 1000 blocks of the differential file table 6-14 holds.

Table 6-14: Duplication rate choosing 1000 numbers from 1..16384

Chosen	Expect
1	969.5
2	15.2
≥ 3	v. few

Whereas for the smaller files almost 10% of blocks would be chosen more than once, for the larger files this figure is less than 2%. The immediate consequence of this will be that there will be more *distinct* blocks in the differential part of the larger file than of the smaller, and hence the overheads of translation will be increased. Further overheads are introduced by false positives from the bitmap filters, since some blocks will now require the translation table to be checked in order to decide that they do not, after all, reside at the level in question. Table 6-15 gives, for 1000 blocks in the differential part and 5000 more randomly-chosen blocks read from the virtual file (the blocks originally written to the static

[1]The only overhead incurred by having "too big" a filter is one of space, both in the disc file and in main store (on a paged machine this might induce extra disc traffic through paging).

part), the expected frequencies of blocks not being in the differential part, of blocks which the filter indicates may be in the differential part but do not actually reside there, and of blocks which are genuinely in the differential part.

Table 6-15: Expected filter performance for large files

Present	Expect
No	4389.6
Maybe	457.8
Yes	152.6

Thus, combining "yes" and "maybe", we see that the bitmap filter would give a positive response for 12% of choices, of which 25% translate to blocks in the differential file ("Yes"), while the remaining 75% are false positives ("Maybe").

Table 6-16: Large base files

	4K	16K
CPU time (milliseconds)	52257.8 (108.9)	52668.9 (99.48)
Elapsed time (seconds)	385.22 (4.287)	457.96 (5.071)
Blocks read	8469.2 (57.96)	8813.6 (55.07)
Blocks written	6177.9 (7.833)	6199.8 (7.438)
Cache hits	7958.6 (149.5)	8190.6 (104.8)
Cache misses	14468.4 (58.04)	14813.3 (55.20)
False positives	0.0 (0.000)	458.08 (11.25)

Total elapsed time for test: 12h 56m 37s

Turning to table 6-16 we see that, for the larger file, the incidence rate of false positives is exactly as expected. This has resulted in increases of 3% in cache hits, 2% in cache misses, 4% in blocks read and less than 1% in both blocks written and in CPU time. On its own the 4% increase in disc I/O would be insufficient to account for the 19% increase in total elapsed time. The increase in size of the virtual file means, however, that whereas for the smaller file only 30 disc cylinders were required, over 100 will be required for the larger file. As blocks are chosen at random, this means that the heads will be on-cylinder less often and so more seeks will be required (the operating system does not issue null seeks). Furthermore, the seeks will be over a larger distance for the larger files, and therefore correspondingly slower.

6.10.1 Large files and disc seeks

In order to ascertain the effect of disc seek time the test was repeated using "ordinary" random access files, which provide the same random access facilities with minimal overheads but without the added flexibility of experiments. The total elapsed time for the small file (averaged over 10 runs) was 285.7 seconds (standard deviation 0.48), while for the large file it was 349.8 seconds (0.88), an increase of 22%. It appears, therefore, that, despite a bitmap filter which is "too small", the extra overheads introduced by the virtual file system are not great, and that the hardware introduces overheads which are at least as large and which would be present whatever software were used.

6.11 Small experiments

It will not always be the case that experiments are as large as the 1000 block updates of the tests described above, and indeed one of the motivations behind the introduction of experiments was the wish to make small scale trial modifications to a design while leaving the original intact. In order to check that the overheads for small scale updates are not excessive, four of the tests (the basic performance test, the cache sizes test, the translation hash-table test and the large files test) were repeated with 100 blocks for the first part, rather than 5000, and 20 blocks for the second part, as opposed to 1000. Note, however, that since we are choosing from the range 1..4096, the static part must be regarded as being at least this size.

A smaller number of blocks will result in fewer blocks being chosen more than once. Table 6-17 gives the expected numbers of blocks, of a total of 100 choices from the range 1..4096, which would be chosen once, twice or more often.

Table 6-17: Result of choosing 100 numbers from 1..4096

Times	Theory
1	97.9
2	1.1
≥ 3	v. few

It will be seen that, whereas for 5000 choices only 32% would be expected to be chosen only once, for 100 choices 98% would be expected to be chosen only once. As before, however, a new block is taken for each choice during the write phase of the test, and the number chosen is still sufficiently larger than the cache size to ensure that the earlier blocks written are flushed out of the cache by the later ones, while the later ones written are flushed out by the first ones being re-read.

Choosing 20 blocks from the range 1..4096 gives rise to table 6-18. It is now the case that duplicate choices are very rare indeed, whereas for 1000 choices almost 20% of the choices would be of duplicate blocks.

Table 6-18: Result of choosing 20 numbers from 1..4096

Times	Theory
1	19.9
≥ 2	v. few

The situation is the same as regards blocks from the first 100 being duplicated in the second 20, as is shown by table 6-19.

Table 6-19: Expected duplication rate of 20 blocks in 100

Present	Expect
No	99.56
Yes	0.44

Whereas with the larger number of choices 20% of the first 5000 blocks chosen would also be chosen in the second 1000, for the smaller number of choices there will be almost no duplicates at all.

The reduction in the number of blocks in the differential file also means that collisions in the hash-table are much less likely. Table 6-20, derived from 100 replications where 20 blocks were written to a differential file and the resulting translation chains dumped, gives the expected numbers of virtual blocks whose translation requires chains to be followed down to the given distance (including the base hash-table entry).

Table 6-20: Overflow chain length (entries)

Distance	Expect
1	19.00
2	0.92
3	0.03
≥ 4	v. few

Very few blocks would be expected to require any overflow chaining at all, and the overheads incurred would be minimal due to the very short lengths of such chains as would exist. The situation for the number of blocks accessed during the translation process is similar, as table 6-21 giving the expected numbers of virtual blocks whose translation would involve accessing the corresponding numbers of translation blocks (including both hash-table and overflow blocks).

This table has, again, been derived from 100 replications of test *C*. Only very few blocks require overflow chaining at all, but even if all 20 collided on the same base entry in the hash-table only one overflow block would be required. This is because each overflow block can hold 42 overflow entries, and hence, in the worst case, all 19 entries required could be

Table 6-21: Overflow chain length (blocks)

Blocks	Expect
1	19.00
2	0.95
≥ 3	none

accommodated on the same block. This explains the "none" in the table above. Since the total number of translation blocks is so small, only one more than the size of the base hash-table at worst, they will tend to remain resident in the cache, and the overheads incurred from this source will be correspondingly low.

6.11.1 Basic performance test

Consider table 6-22. One difference which arises with the larger scale test is that whereas with 1000 modified blocks the earlier written ones were flushed out of the cache by the later written, which were, in turn, flushed out by the earlier blocks being re-read, the blocks being written in the second part of the test are sufficiently few in number that they can all remain resident during the second part of test *A*. Hence, although all 120 blocks are written out, including some administrative ones (the cache is, of course, flushed before the file is closed), the second 20 do not require to be re-read. This is not the case for test *C*, where the re-reading of the first 100 causes the cache to be flushed, together with their translation blocks, and will require to be re-read later. For test *B* the 20 blocks of the second part, together with their translation blocks will total around 24 or 25 in number, depending on whether there are any duplicated choices (although rare, these do occur occasionally in practice) and hash-table collisions. If either a duplicate choice occurs or there are no collisions (and hence no overflow block is required) then the total number of blocks will be 24. This number is exactly the same as the cache size used for the tests, and so all blocks will remain resident, as was the case in test *A*. If there are no duplicate choices and at least one collision then the total number of blocks will be 25. This is one greater than the size of the cache, and hence the first block written will be flushed out by the last, the second written will be flushed by the first being re-read, the third written will be flushed by the second being re-read, and so on. The comparatively high standard deviation for the blocks read is caused by this behaviour, which was quite evident from the original complete printout of the results of the experiment, a distinctly bi-modal distribution being observed.

As was the case in the larger scale test, the overheads for test *C* are greater than those of test *B*, which are, in turn, greater than those of test *A*. There is very little difference in total elapsed time for the tests, and it is doubtful, in any case, whether such differences as do exist are large enough to be perceptible to the user, particularly on a time-sharing system.

The time to perform a merge is now about 1.4 seconds (elapsed time). This should be compared against the cost of copying the database which, since we are choosing blocks from the range 1..4096 must be at least 4096 blocks in size. This cost is, as before (page 61),

Table 6-22: Basic performance test (fewer updates)

	Test *A*	Test *B*	Test C	Test *M*
CPU time (milliseconds)	904.72 (4.185)	956.42 (12.53)	1013.38 (5.518)	218.66 (4.992)
Elapsed time (seconds)	7.12 (0.385)	7.28 (0.454)	8.10 (0.416)	1.38 (0.490)
Blocks read	100.46 (0.788)	113.36 (8.378)	128.32 (1.096)	27.56 (0.541)
Blocks written	122.0 (0.000)	131.66 (0.487)	131.62 (0.490)	22.94 (0.240)
Cache hits	20.54 (0.788)	37.24 (1.506)	33.10 (1.199)	164.26 (0.527)
Cache misses	99.46 (0.788)	121.10 (16.61)	247.80 (1.702)	4.62 (0.490)

Total elapsed time: 20m 24s

160 seconds. The small overheads caused by using virtual files pale into insignificance beside this figure, the comparison only serving to emphasise how cheap virtual files really are.

6.11.2 Cache sizes

Table 6-23 summarises the results obtained by varying the cache sizes to 8, 16, 24, 32 and 40 blocks as before. There is some slight evidence to suggest that with an 8 block cache there is more disc traffic than with the larger caches, due to the translation blocks being flushed out. Apart from that, all the results are practically identical except for CPU time, which has increased with the extra processing required to search a larger cache. Such differences as may exist in the total elapsed times are totally swamped by the inherent variability caused by the random number generator, and there is no evidence to support any suggestion that it has been affected by the changes in cache size.

6.11.3 Cache disciplines

Table 6-24 shows the result of varying the cache discipline. As before, the "toss immediate" algorithm shows superior performance to the "least recently used" algorithm, although as the number of translation blocks is not so large the resulting benefit is correspondingly reduced. Nevertheless, it remains the case that the user software can effect an improvement in performance by making use of information not available to the virtual file manager.

Table 6-23: Different cache sizes (fewer updates)

	8	16	24	32	40
CPU time (milliseconds)	914.64 (2.731)	961.06 (2.714)	1008.90 (4.643)	1048.46 (7.229)	1115.42 (3.011)
Elapsed time (seconds)	7.78 (0.465)	7.82 (0.388)	7.86 (0.535)	7.82 (0.482)	8.02 (0.428)
Blocks read	135.78 (2.659)	129.20 (1.294)	128.50 (1.182)	128.10 (1.249)	127.66 (1.222)
Blocks written	135.50 (1.693)	132.06 (0.767)	131.60 (0.495)	131.52 (0.544)	131.54 (0.579)
Cache hits	25.84 (2.721)	33.00 (1.917)	33.50 (1.581)	33.94 (1.953)	33.68 (1.634)
Cache misses	255.78 (2.659)	249.18 (1.304)	248.48 (1.216)	248.02 (1.301)	247.62 (1.260)

Total elapsed time: 33m 25s

Table 6-24: Cache disciplines (fewer updates)

	LRU	Toss
CPU time (milliseconds)	1245.54 (2.242)	1307.60 (6.484)
Elapsed time (seconds)	8.24 (0.431)	7.32 (0.471)
Blocks read	128.28 (0.904)	111.26 (2.522)
Blocks written	131.52 (0.505)	120.06 (2.645)
Cache hits	33.18 (1.480)	289.22 (2.985)
Cache misses	247.72 (1.471)	230.02 (2.591)

Total elapsed time for test: 13m 14s

6.11.4 Hash-table size

Again it is the case that changing the size of the translation hash-table has made very little difference to the performance of the system. It is, however, the case that a larger hash-table consumes a larger proportion of both total file size and disc transfers, with the reduction in density of entries which results meaning that individual blocks are accessed less frequently and may become non-resident in consequence. This effect can be seen in table

6-25, where increasing the hash-table size has resulted in an increase in cache misses and a corresponding decrease in cache hits, the overall effect being to increase disc traffic and CPU time. The resulting increase in total elapsed time is so close to the background noise of the random number generator that it may not be a real effect, and would probably not be noticed by the user. Given that there was no observable difference in performance for the larger scale tests, it would seem sensible to use a fairly small translation hash-table.

Table 6-25: Translation hash-table sizes (fewer updates)

	4	8	12	16
CPU time (milliseconds)	1246.54 (2.549)	1252.18 (3.001)	1256.88 (2.685)	1260.74 (2.856)
Elapsed time (seconds)	8.22 (0.418)	8.32 (0.471)	8.72 (0.454)	8.80 (0.404)
Blocks read	128.14 (1.010)	135.22 (2.013)	140.44 (2.628)	144.18 (3.361)
Blocks written	131.48 (0.544)	138.92 (0.922)	145.46 (1.280)	151.28 (1.642)
Cache hits	33.40 (1.895)	25.60 (1.818)	20.22 (2.589)	16.38 (3.410)
Cache misses	248.10 (1.055)	255.18 (2.007)	260.46 (2.628)	264.18 (3.361)

Total elapsed time: 28m 58s

6.11.5 Large files

For virtual files larger than 4096 blocks the possibility of false positives again arises. These should be many fewer than with the larger scale of updates, however, as is shown by table 6-26, which gives the expected frequencies, from 100 blocks, of blocks which the filter indicates are not in a differential file of 20 updated blocks, those which may be in the differential file but turn out not to be, and those which genuinely are in the differential file.

Table 6-26: Expected duplication rate for large files

Present	Expect
No	99.757
Maybe	0.181
Yes	0.062

The figures in the table are derived from 1000 replications. False positives would be expected to be infrequent, and duplicate choices very rare indeed.

Turning to table 6-27 we see that the actual rate of false positives is low, as expected, and

Table 6-27: Large base files (fewer updates)

	4K	16K
CPU time (milliseconds)	1009.60 (4.891)	1009.24 (4.538)
Elapsed time (seconds)	7.86 (0.405)	9.24 (0.431)
Blocks read	128.28 (1.230)	128.74 (0.828)
Blocks written	131.62 (0.490)	131.56 (0.501)
Cache hits	33.70 (1.972)	32.60 (1.107)
Cache misses	248.28 (1.230)	248.74 (0.828)
False positives	0.0 (0.000)	0.28 (0.536)

Total elapsed time: 14m 23s

that there is no evidence of any differences in any of CPU time, blocks read, blocks written, cache hits and cache misses, such variations as there are being well below the noise threshold. Despite this, the total elapsed time has increased by 18%. As CPU time and disc I/O are the same for both smaller and larger files, the only cause of this difference can be increased seek time. The test was repeated using "ordinary" random access files and, with identical processing being carried out, the results for the 4K file was 6.0 seconds (standard deviation 0.00), while for the 16K file it was 7.4 seconds (0.52), an increase of 23%. As with the larger scale of update (section 6.10.1 on page 72), any increase in system overheads caused by using larger files are more than swamped by hardware effects.

6.12 Differential files and shadowing compared

We predicted in section 3.6 that a differential files implementation would perform no worse than a shadowing implementation. In order to test this experimentally, a system based on shadowing was written with an interface presented to the user which was identical to that of the differential files implementation[1] and interfaced to the host filing system at the level of the disc cache, with the result that as much code as possible was common to the two implementations. Note that the shadowing system implemented transactions only, using the same algorithm as the *Shrines* utility (sections 3.5 and 7.2, and appendix C); a full implementation of experiments was not done. Hence any inferiority in performance of the

[1]Apart from the procedure names so that the two could be linked into the same test program

shadowing implementation of transactions would also be apparent in a full implementation of experiments, only to a greater extent. Tests were carried out on the *3220* with only one of the store boards present to avoid biasing of the results by having one technique loaded into the faster store board with the other in the slower. The results are given in table 6-28.

Table 6-28: Differential files and shadowing

	Diff.	Shadow.
CPU time (milliseconds)	40210.6 (64.55)	28115.8 (68.57)
Elapsed time (seconds)	356.78 (4.871)	489.76 (1.585)
Blocks read	8474.4 (59.57)	13611.3 (48.89)
Blocks written	6178.1 (6.880)	7766.2 (16.16)

Total elapsed time for test: 11h 59m 35s

As we predicted, the shadowing implementation has performed less well than the differential files implementation. Despite using less CPU time it has required 27% more elapsed time by virtue of the 45% increase in disc traffic engendered by the much larger translation data structures. Although it is true that the access pattern employed in these tests is much less localised than would be expected in a "real life" situation where the effect of the cache and the processing performed by higher levels of the system would result in a lesser relative advantage to a differential files system, nevertheless the results obtained, together with the operational advantages engendered by the filing system, confirm that such a system is superior to a shadowing system. Indeed, the case of a bare disc, mentioned in section 3.5, would be most sensibly approached in two stages:

1. implement an extent-based filing system

2. implement virtual files based on differential files

6.13 Summary

We have established that the overheads incurred in using a virtual file system based on differential files are acceptably low, and would become proportionally less as the amount of processing on each block increased (none was performed during the above tests). For small quantities of updates the extra overheads incurred are so small that the user would almost certainly notice no degradation in performance, and on most machines such changes as there were would be sufficiently less than the variations in perceived performance caused by factors beyond the control of the user, such as competition from other users, that they may be ignored.

Chapter Seven
Virtual Memory Systems

The benefits of virtual memory systems are too well known to require much comment here (see, for example, [Denning 70], or any operating systems textbook). Quite apart from the advantages for the memory manager of the operating system, and the removal, or at least relaxation, of address space constraints on the user, there is the added convenience that the complex manipulation of I/O buffers may be eliminated by creating a logical equivalence between disc files and virtual memory; the user is then able to access his files as though they were "ordinary" portions of store using the "ordinary" machine instructions provided for this purpose, all "input" and "output" going via this mechanism. It may even be the case that terminal I/O is routed via such a mechanism, as is the case with *EMAS* [ERCC 78, ERCC 82], though here it is packaged up so as to give the appearance of a serial byte-stream. The result of this is that it is much easier to write data-handling programs, as both the absence of a complex buffer manager, a potential source of a number of obscure bugs, and the conceptual simplicity of being able to act on the entire file rather than a small window onto it, lead to clearer and more correct programs.

One impediment to the obvious extension of this concept, *viz* directly mapping an entire database, is that databases are generally large, perhaps so large that they cannot fit into a virtual address space [Stonebraker 81]. In general, however, a transaction will only access a small part of such a large database, and so this problem may be overcome by only mapping "interesting" parts of the database into virtual memory, as was done with *System-R* [Traiger 82], for example. Unlike the windows provided by a buffer manager, however, these windows will guarantee that, for the duration of the experiment, accessing a particular virtual address will always give access to the same database address.

Furthermore, there may be difficulties in guaranteeing atomic transactions using mapped files, since it must be possible both to force pages out to disc and to lock pages into main memory so that they will not be written out. These problems are discussed in [Grimson 80], who was forced to use a before-image log in order to implement transactions.

7.1 Databases and virtual memory

The interaction between the database system and the virtual memory system may not be entirely straightforward in either a "conventional" system with a buffer manager or in a directly mapped system (see [Sherman 76, Brice 77, Lang 77, Fernandez 78, Traiger 82]). In addition to the page replacement algorithm used, which may not be optimal for the pattern of access to the database,[1] there is also the question of more subtle trade-offs between I/O costs to the page file and to the database disc file. Three cases are presented below:[2]

- The database manager maintains a small cache in virtual memory. Blocks are explicitly transferred to and from backing store by the operating system at the request of the database manager. The cache will be sufficiently small and accessed sufficiently frequently that it is sensible to lock it into the user's working set. This corresponds to the block-based implementation described in chapter 5.

- The database manager maintains a large cache in virtual memory. As with the previous case, transfers to and from disc take place at the request of the database manager. Now, however, it is not sensible to lock the cache into the user's working set, as this would give rise to excessive paging of the remaining portion of the working set; instead the cache is paged along with the rest of the user's virtual address space.

- The entire database (or that part of it which is considered "interesting") is mapped into the user's virtual address space, with transfers to and from the disc being handled by the host operating system's page fault manager. This approach is used in the *Shrines* system described below (section 7.2).

At first sight there would appear to be little to choose between the first two of these schemes, except perhaps that the former might be expected to use less CPU time than the latter due to having a smaller cache to search, since in the former scheme blocks are explicitly transferred to and from disc by the database manager, whereas in the latter scheme blocks are implicitly transferred via the paging mechanism for the most part, with only the initial read from the database and the final write to the database being explicit, thus merely substituting one form of disc I/O operation for another. It may, however, be the case that one form of transfer is more expensive to perform than the other: for example, head movement, particularly that caused by fragmented files and competing users, may result in the cost of "do-it-yourself" I/O operations being higher than that for corresponding I/O operations to the page file. CPU time is unlikely to be a dominant factor here, since the

[1]If the operating system allows processes to selectively flush their working sets then this may be alleviated to some extent; however there is still the case where the database manager knows that a particular block will be required but is unable to ask the operating system to prefetch it.

[2]It is assumed that transactions and/or experiments are being implemented by redirecting a newly-modified block to a new site on disc, as is the case with differential files or shadowing.

necessary checks on the D.I.Y. I/O will be quick to perform compared to the elapsed time waiting for the disc; indeed, if the virtual file system is trusted by the operating system, as is reasonable for a facility of this nature, then the checks may be only vestigial. To counterbalance this, the latter scheme may result in extra unnecessary disc transfers taking place as the cache is flushed during the closedown sequence, since it may be necessary to page in part of the cache in order that the contents may be written out to the database file; these two transfers would not be incurred with the former scheme, as the modified page would have been written out to its final site on disc rather than to an intermediate one in the page file. The trade-off between these two approaches is, of course, application and system dependent: if the cost of an explicit disc transfer is high compared to that of an implicit one, or if the transaction is preponderantly read-oriented, thereby incurring fewer unnecessary closedown transfers, then the balance is tipped towards the latter scheme; whereas if the costs of the two forms of transfers are similar or there is quite a high density of updates performed by the transaction then the balance will be tipped towards the former scheme. The latter scheme was not practicable on the *3220*, the non-paged nature of the memory management hardware dictating that a small resident cache be used. Furthermore, the latter scheme may result in more page faults, as the total process virtual address space size will be larger than that of the former scheme; this effect may be ameliorated to some extent, however, if the operating system allows the user program to purge parts of its working set which it knows it will not want to use again in the immediate future (this corresponds to the "toss immediate" strategy described in section 6.7).

The third of these schemes, involving mapping the database into virtual memory, has features in common with both the other schemes, and thus inherits advantages and disadvantages of both. It shares with the large-buffer scheme the fact that the normal operating system page-replacement strategy is used to determine which part of the database is resident in main memory at any particular time, with possible non-optimal paging behaviour ensuing. All disc transfers take place directly to and from the database disc file, as in the small-buffer scheme, and hence page faults incurred during the copying-back phase are avoided, although there may be an extra I/O cost incurred through not using "normal" virtual memory. Furthermore, address validation is performed by the hardware, with the consequent saving in CPU time and programming convenience of avoiding the large number of explicit software checks which both the other schemes must employ. The situation is complicated even more if the facilities provided are not quite ideally suited for the purpose to which they are being put, since a contorted method of setting up the appropriate mapping, or over-enthusiasm by the host operating system, can greatly magnify the overheads of this method. It may be, however, that the convenience factor of being able to access the database directly, rather than indirectly via a buffer manager, will tip the balance in favour of the mapping scheme at the expense of one of the buffer schemes, even if the performance overheads of the former are greater than those of the latter.

The approaches outlined above will be further discussed in section 7.3 below in the context of *VAX/VMS*.

7.2 Shrines – virtual files for VAX/VMS

Although the normal way of considering files under *VAX/VMS* is as a collection of records, accessed via the *Record Management Services* [DEC 82a], nevertheless it is possible to access files via the memory management facilities in much the same way as for *EMAS*, with the additional feature that pages from a *section file* (as a file mapped in this way is known [DEC 82c]) may be mapped into virtual memory in an arbitrary order rather than necessarily being mapped sequentially into a contiguous area as is the case for *EMAS*. Together with the memory protection and exception handling mechanisms, this allows the construction of a facility whereby virtual files may be mapped into virtual memory, with all the convenience which this provides. The *Shrines* system, conceived and part-implemented by Paul MacLellan to provide secure transactions on mapped files, provides such a facility.

To the end user, a shrine appears similar to any other file which would be mapped as a global section, *viz* as a (contiguous) area of store, the contents of which may be preserved in a disc file when the shrine is checkpointed. Shadowing, with two root blocks and a two-level map (section 3.5 on page 34) is used to enable any updates to the shrine to be abandoned, either voluntarily by the program or involuntarily by a system malfunction. The user interface is described in appendix C.

When the shrine is opened the virtual data pages are mapped into a contiguous area of the user's P0 virtual address space with the protection set to *user*-read. When the applications program wants to read data from the shrine it merely uses the normal *VAX-11* instruction set, with any necessary transfers from backing store being performed by the page fault handler in the *VMS* kernel. The first time the program attempts to write to a virtual page it will incur an access violation fault; this is fielded by an exception handler established by the shrine manager using the secondary exception vector; the exception handler saves the contents of the virtual page on the *user*-mode stack, unmaps the old page, allocates a new site in the shrine section file and updates the translation map, maps the new virtual page at the same virtual address, restores the contents of the virtual page from the stack, and finally returns control to the exception dispatcher indicating that the faulting instruction should be restarted. Note that the exception handler may be called recursively when the translation map is updated, as the same technique is used to detect first writes to the map pages as is used to detect first writes to the data pages, the second-level map being itself mapped via the top-level map in the root blocks.

At present the entire shrine manager executes in *user* mode. This is clearly undesirable, however, since the map pages and current root page should be mapped in such a way that they may not be corrupted by an incorrect applications program. In order to remove this possibility it is intended to use the change-mode mechanism provided by privileged shareable images [DEC 82d, DEC 81] to allow the shrine manager to execute in *exec* mode with its own data structures protected against *user*-mode writes. This has the further advantage that the pages of the shrine would be owned by *exec* mode, rather than *user* mode, with the result that *user*-mode code would be unable to affect the integrity of the shrine by altering its protection.

7.3 Virtual files and virtual memory: the overheads

Let us consider further the relative merits of the three methods described in section 7.1 above in the context of the *VAX/VMS* system. The conclusions we draw will not, of course, necessarily hold for other machines and operating systems, though it is of interest to see how the factors indentified above may affect the design of a virtual file system. The various options were evaluated by reading and writing randomly chosen blocks from a 4096-block contiguous-best-try[1] file, both by treating it as a section file with the page fault mechanism transferring the pages as required (the third method described on page 81), and by performing QIOs directly to the disc driver (the first two methods). As the second method involves buffering blocks in paged virtual memory the costs of reading and writing blocks from the paging file were also obtained. In the case of the section file tests, the page fault cluster size for the section was set to one, to correspond more closely with the internal fragmentation of the disc file caused by the shadowing approach of the shrine manager. In all cases the relevant process and system parameters affecting paging performance were as shown in table 7-1.[2] Tests were run late at night when the system was quiet, so the test process was able to borrow pages up to its full WSExtent; the first two working-set parameters are therefore of much less relevance than the third.

Table 7-1: System parameter settings for virtual memory tests

```
! Process working-set parameter values
WSDefault      = 128
WSQuota        = 384
WSExtent       = 768

! SysGen paging parameter values
BorrowLim      = 300
FreeGoal       = 192
FreeLim        =  64
GrowLim        =  63
MPW_HiLimit    = 500
MPW_LoLimit    = 120
MPW_Thresh     = 200
MPW_WaitLimit  = 500
MPW_WrtCluster =  96
PFCDefault     =  64
```

The results obtained are summarised in the graphs on pages 86 to 90, where the total elapsed time for the reading or writing part of the experiment is plotted against the number of accesses to the "database". Note that initialisation and closedown costs are not included

[1]Placed by the filing system so as to be as "contiguous as possible"

[2]A brief description of the *VAX/VMS* memory management system and the parameters which affect its operation can be found in appendix D. See also [DEC 81, DEC 82e].

in the measurements. Linear regression lines, constrained to pass through the origin on the principle that doing nothing must cost nothing, were fitted to all the classes of measurement. The slopes of these lines indicate the cost (in elapsed time) per page fault (or QIO request) assuming that this is constant over the number of blocks accessed (as will be seen, this assumption is not valid in all cases).

Figure 7-1 on page 86 shows the relative costs of reading and writing random blocks from a section file and writing to the system page file. It will be seen that the cost of reading from the section file does not increase linearly with the number of pages accessed. This is because when a page which has been read in from the section file is removed from the process's working set it is placed on the system-wide free list with the possibility of being recaptured should there be a subsequent fault for that page. If the system is lightly loaded, as was the case for all the test runs described here, it is possible for the entire section file to become resident after which page faults will be almost free. It will be recalled from chapter 6 that choosing random numbers from a restricted range results in duplicated choices, with proportionally more duplications occurring for more choices. Hence, although it will probably not be the case that the entire file has become resident, nevertheless there will be sufficient duplicated choices that page recaptures play a significant role in reducing the proportional cost per page fault for larger numbers of accesses.

Figures 7-2 and 7-3 on pages 87 and 88 show the costs of writing random pages to a section file and to the page file respectively. The shape of the two graphs is very similar, showing that the same mechanism is responsible for each, although the slope of the section file graph is much steeper than that of the page file graph. There are two mechanisms in operation here: the page recapture mechanism described above applies, so that once a page has been read in it is likely (assuming that the system is lightly loaded) that it will not need to be re-read; furthermore the system does not write out "dirty" pages immediately they are removed from a process's working set, but rather places them onto a modified page list, with the Swapper writing them out only when the list grows above MPW_HiLimit pages in size. The overall effect is that with fewer than about 1250 accesses[1] the number of pages being modified does not cause the modified page list to fill up and hence does not cause any pages to be written,[2] with the result that the shape of the graph corresponds to that obtained by reading from the section file but not writing. Above this threshold the modified page list will fill up, and the process will have to wait until it has been written before proceeding. The pages thus written will, however, be placed on the free list, from which they can be recaptured, and hence above the threshold the shape of the graph is a combination of that obtained by reading from the section file with that obtained by writing blocks to the disc (see below).

[1]The sum of the process's working set size, limited by the WSExtent parameter, and the maximum size of the modified page list, defined by the MPW_HiLimit parameter, *viz* 768 + 500.

[2]A very pronounced pause was observed when the section file was closed as the modified pages were flushed. This does not contribute to the measurements, however.

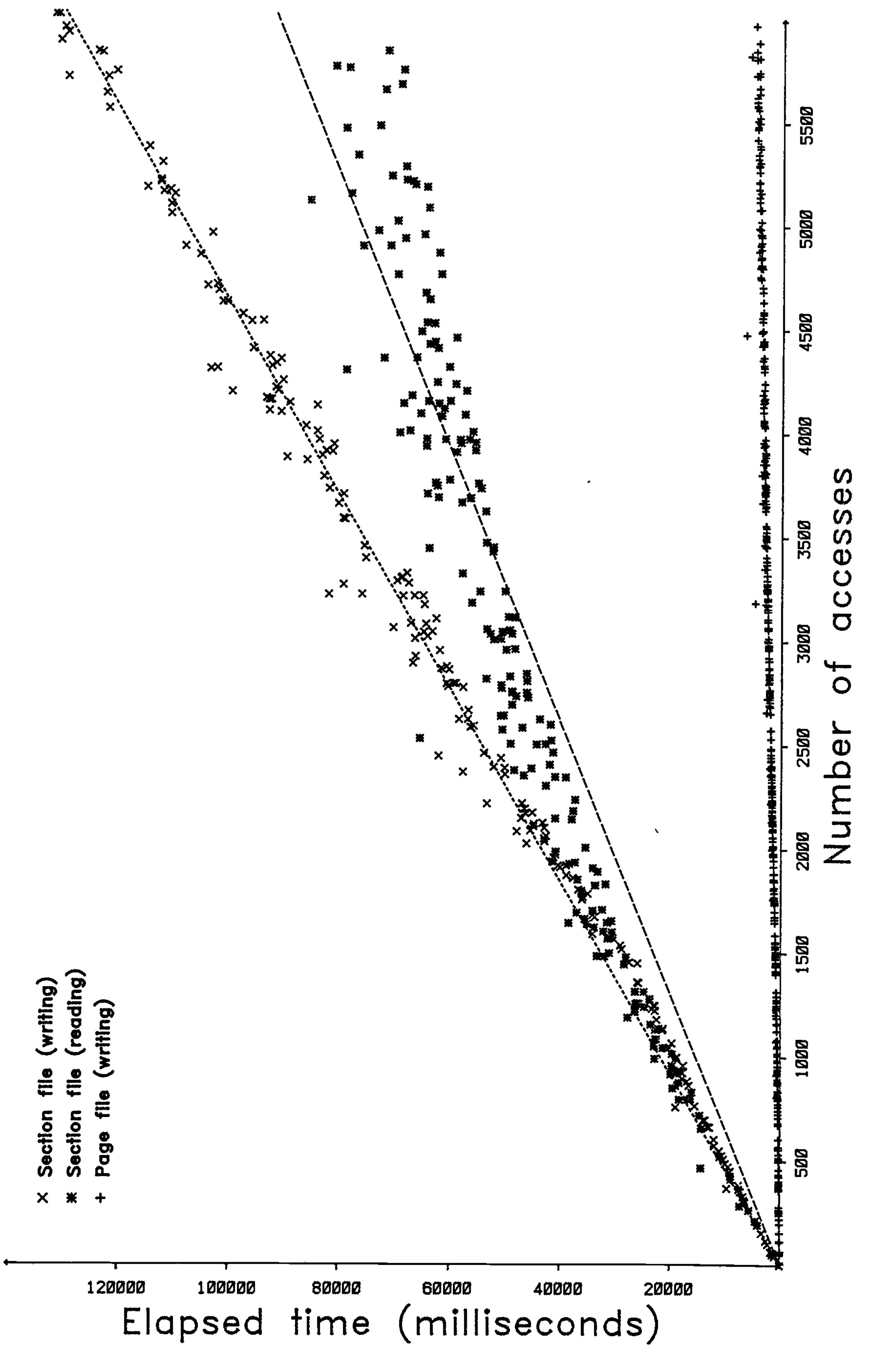


Figure 7—1: Section file versus page file

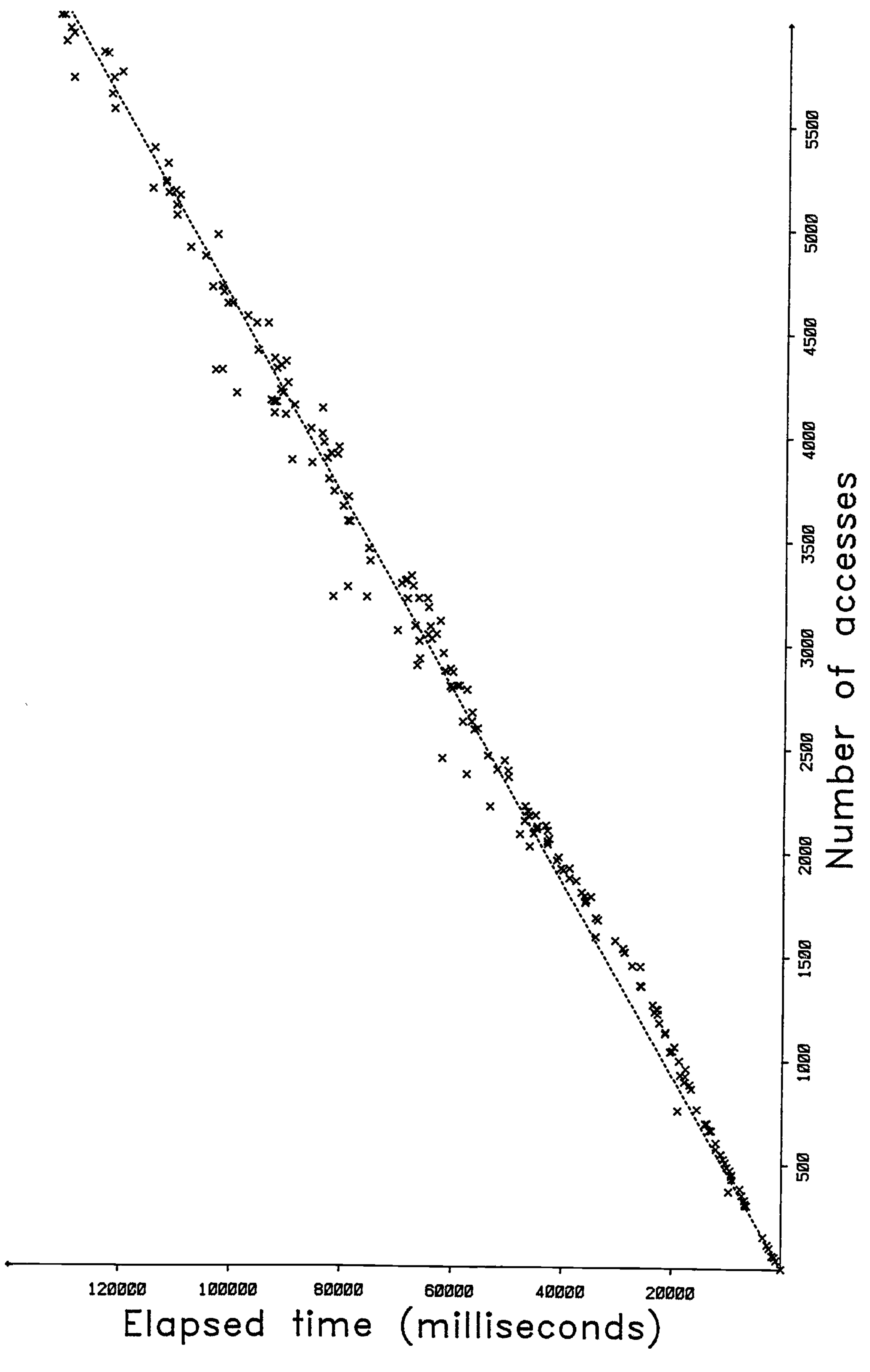


Figure 7–2: Section file (writing)

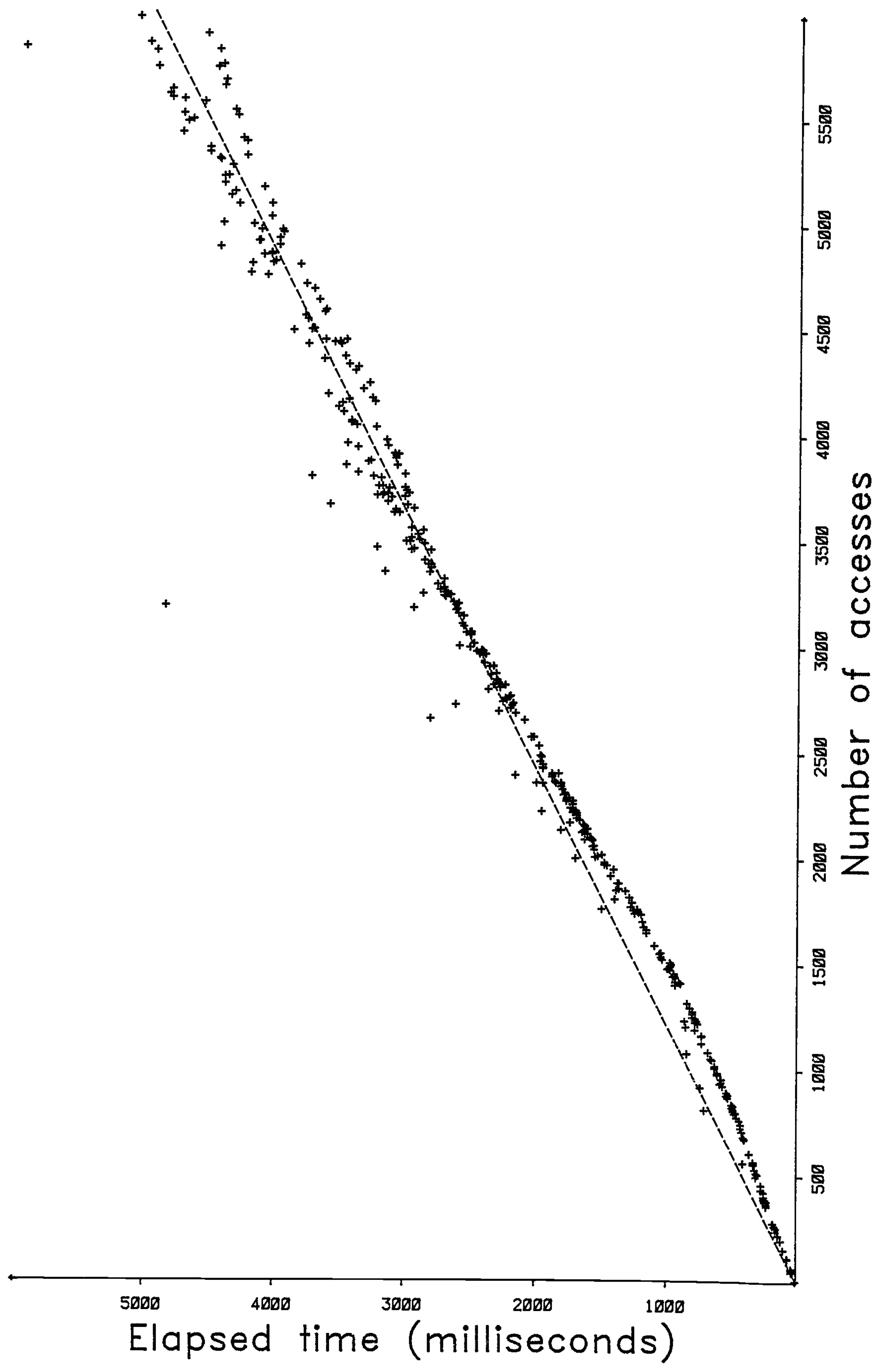


Figure 7–3: Page file (writing)

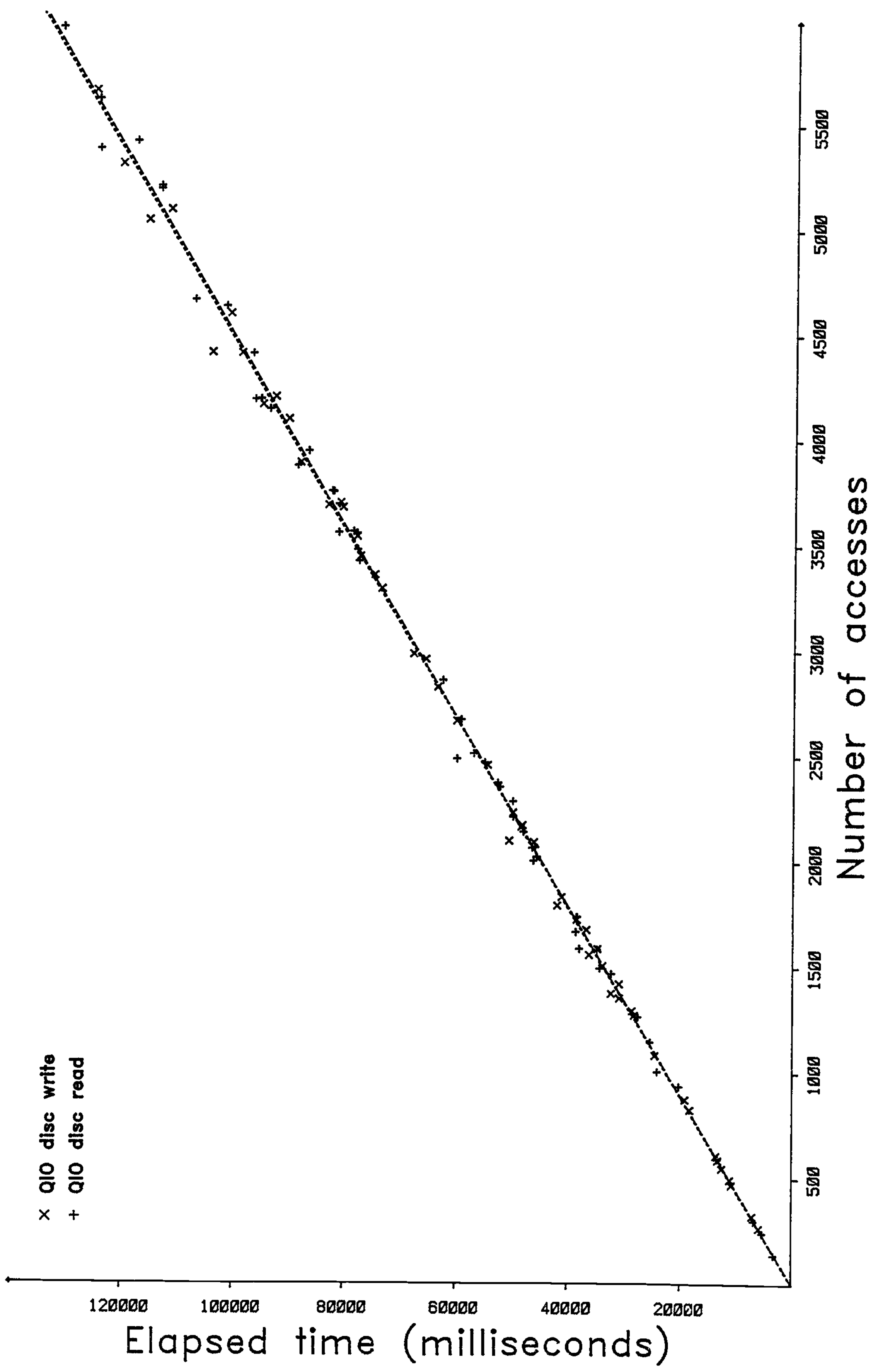


Figure 7–4: QIO reads and writes

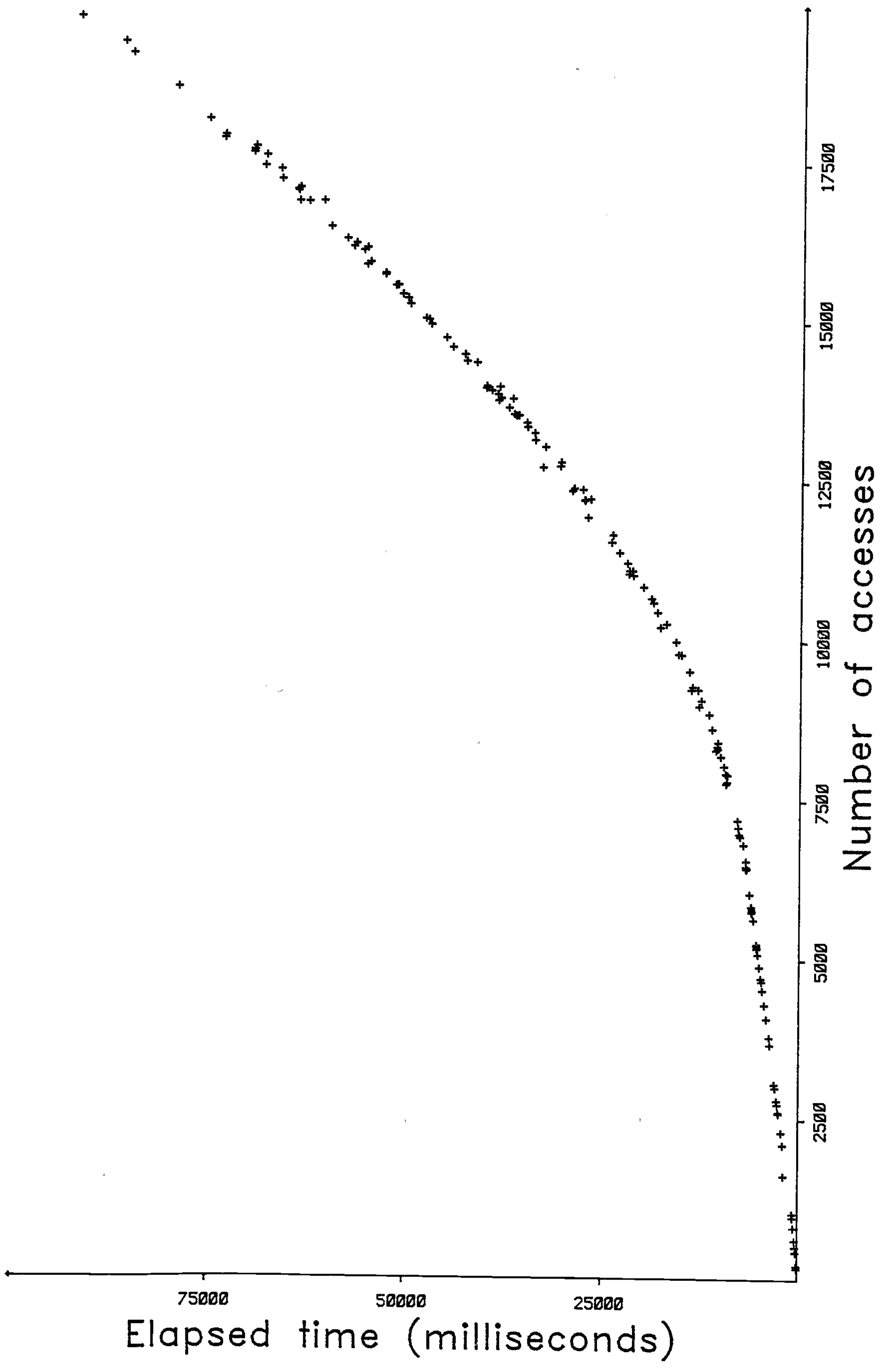


Figure 7–5: Large databases

It will be seen from figure 7-1 on page 86 and table 7-2 on page 91 that the slope of the line for writing to the section file is very much steeper than that for writing to the page file. This is because the Swapper clusters writes to the page file, thus reducing overall disc latency by transferring a number of blocks in one request; this is not done for section files, however, with each block being transferred by a separate request. The page file write cluster size is determined by the MPW_WrtCluster parameter, the value in effect during the tests being 96.

Figure 7-4 on page 89 shows the cost of performing explicit QIO requests to transfer a single block to or from disc. As would be expected, the cost is the same for both reads and writes, this cost being marginally greater than that incurred by writing pages to a section file.

Of course, if the number of pages accessed is larger than the size of the physical memory attached to the processor (4 Mbytes, or 8192 pages, in the case of the tests described here) then it will be the case that recapturing will be impossible for all pages, as their physical slots will have been reused for other pages. This effect is apparent in figure 7-5 on page 90, which shows how the (elapsed time) cost varies with the number of pages accessed.[1] It should be emphasised that it is the number of distinct pages accessed, not the overall size of the database, which gives rise to this effect. In practice, few applications would require ready access to such a large part of a database; for those which do, the only solution is to attach more physical memory to the processor.

The costs (in milliseconds elapsed time) of each of the methods are summarised in table 7-2. This assumes that the total cost increases linearly with the number of accesses, an assumption which is not true for reads from a section file (see figure 7-1), where the cost will be much the same as that of a write to a section file on the first access, but virtually free thereafter due to recapturing.

Table 7-2: Costs per page access/QIO request

Method	Cost
Page file (write)	0.82
Section write	21.68
Section read	15.31
QIO write	22.42
QIO read	22.34

Let us consider the three schemes of section 7.1 in the context of these measurements. The small buffer scheme would be expected to perform worst of the three since it is unable to use recapture techniques to reduce its disc traffic while both of the others can. It also suffers from the additional overhead of requiring an explicit cache search on each block access, a process which is performed by the hardware for the directly-mapped scheme. Of

[1]The graph shows accesses to the page file; the same effect would be observed with a section file, however.

the large buffer and mapping methods, the balance between them depends very much on the pattern of access to the database: if a given page is written out to disc three or more times because the database system is unable to optimise accesses to it, then the large buffer scheme will work out cheaper; if, however, a page is only written out to disc once or twice then the direct mapping scheme will work out cheaper.[1] If w is the number of times a given page is written out, then the cost of the large buffer scheme is

$$22.33 + 0.82\,w + 22.42,$$

while for the mapped scheme it is

$$21.68\,w,$$

with the crossover point being at $w = 2.14$. For reads from the database we would expect little difference between the two methods, since in both cases an initial single-block read is required to bring the page into main store, with the page thereafter being a candidate for recapture. The page file page will be subject to an extra disc transfer since it has been written to by the QIO read transfer while the mapped page is regarded as not having been written to. Typically on the Computer Science Department's VAX with 20 active users the size of the free list is around 2000 pages.

The idea behind performing clustered page reads is that, by transferring a number of pages, the average disc latency overhead per page transferred may be reduced. This will help make practicable such operations as geometric modelling [Eastman 80], where it is essential that there be little delay between fetching individual objects from disc. The assumption is that any pages placed on the free list as a result will be recaptured when they are required rather than incurring the overhead of being faulted individually from disc, the clustered read giving a form of prefetching. In the case of a shrine, the fragmentation introduced in the section file by the LP-map will mean that, in general, clustered reads will not be possible: hence in order to simulate as closely as possible these conditions in the above tests the page fault cluster size (PFC) was set to 1. If, however, a different approach to implementing transactions were employed it might be possible to make use of clustering; a differential files approach, for example, would have contiguous virtual blocks physically contiguous in the lowest level file. In order to ascertain the effect which clustering might have a further set of tests was performed, the results of which are shown in figure 7-6 on page 93. As shown in the legend, six combinations of page fault cluster, reading or writing, and random or sequential access were used.

As would be expected, the costs of performing sequential reads and writes increases linearly with the number of accesses, since as pages are accessed only once there can be no recapture effects. The cost of random reading with the default page fault cluster size increases steeply at first, but levels off at around 500 accesses; by this point most of the pages in the file will have been read in, recapturing accounting for most of the subsequent

[1]This assumes that the system is not heavily loaded, so that the conditions are similar to those at the time of the tests described above. On a heavily loaded system, as was observed while running the test programs interactively during development, there is so much variability in the results that, statistically speaking, very little can be deduced from them.

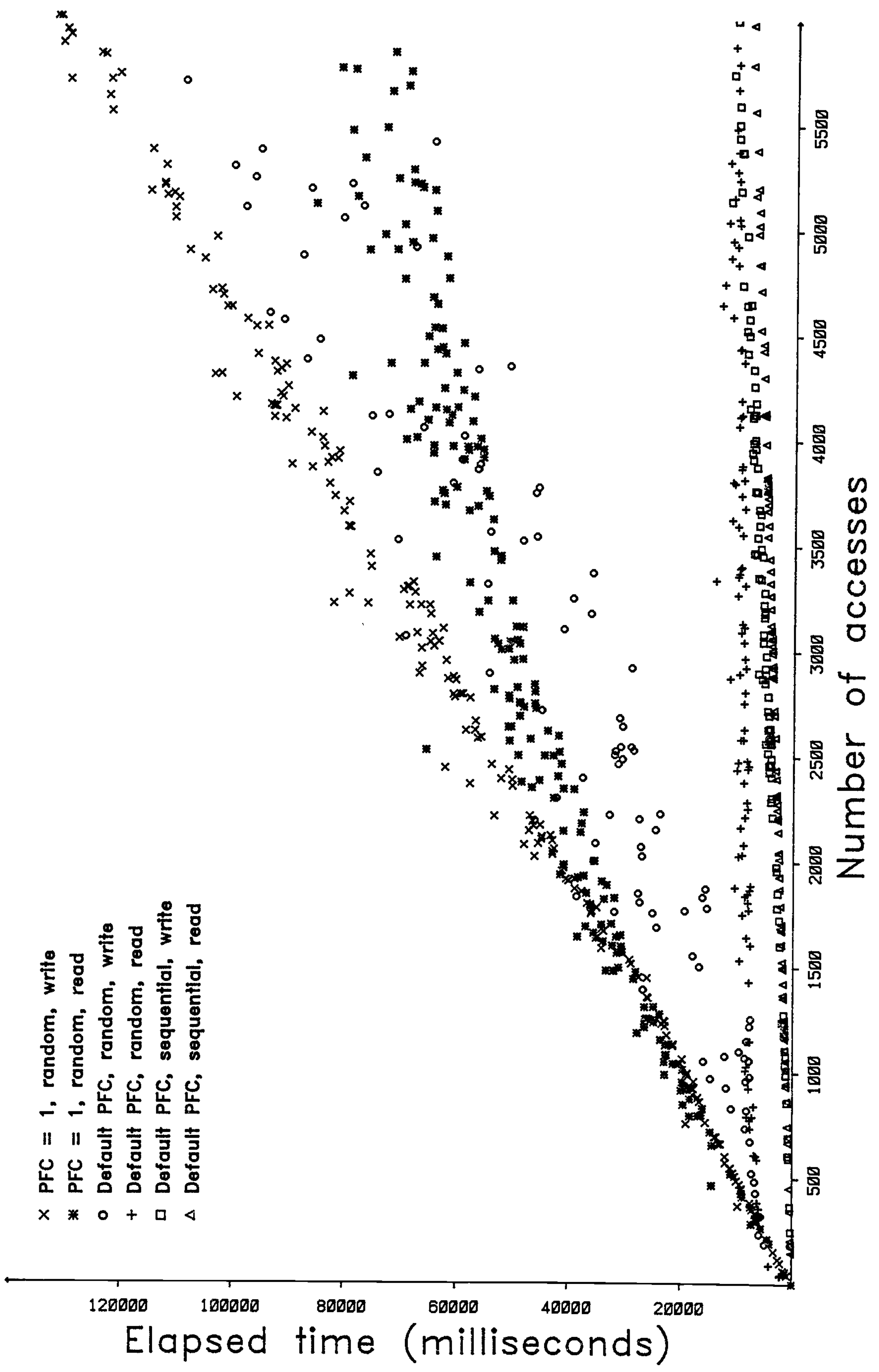


Figure 7–6: Cluster size and sequentiality

"faults" of pages not currently in the working set. The cost of random writes with the default page fault cluster shows a great deal of scatter, though it is, on the whole, less than that for random writes with a page fault cluster size of 1.

It should be remembered that these results were obtained on a lightly loaded system; on a heavily loaded system, with the modified list being frequently written out and the free list being rapidly consumed, there would come a point where reading in "unnecessary" pages could result in "necessary" pages being written out and then immediately read back in again.

With the behaviour seen in figure 7-6, a mapped file would have quite a clear performance advantage over the large buffer scheme provided that the clustering effect could be put to good use. A differential file scheme would allow this, and in addition would incur lower startup costs than does the *Shrines* system using shadowing. The *Shrines* system must map each page in the entire database separately, via the LP-map. This involves a large number of calls to the $MGBLSC system service, with the name of the section specified as an ASCII string in each. The resulting overheads cause quite a noticeable delay when opening a shrine. Using differential files, this overhead could be greatly reduced, since the (lowest) static part could be mapped as a single operation, with individual mapping operations being required only for the differential part(s).

There is one advantage which the buffer schemes enjoy over the directly-mapped scheme: this is that it is much easier to extend the database. With the mapping system, a fixed area of the process's virtual address space is set aside for the virtual file, with subsequent addresses potentially being used for some other purpose. In order to extend such a virtual file it is necessary to unmap/delete the pages and remap/recreate them in a larger hole elsewhere. The only alternative to this on *VAX/VMS* is to allocate extra space for the virtual file to grow. These solutions are either inconvenient or wasteful of the process's virtual address space quota. On a machine with segments as well as pages the necessary hole could be provided without much overhead, but on non-segmented machines this remains a problem.

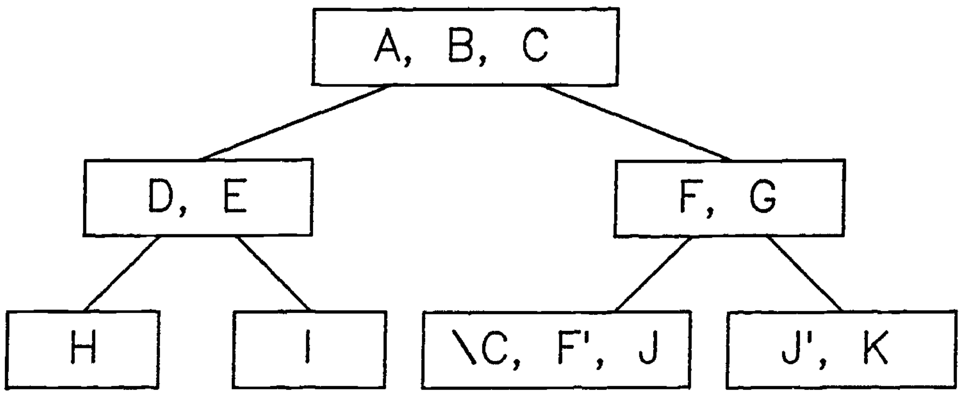
7.4 Shrines - extension to experiments

As we saw above, the *Shrines* system implements transactions using shadowing with a two-level map. Although perfectly adequate for the purpose for which it was originally intended, it does not support experiments. We must therefore ask how the system could be extended.

We find that we run up against much the same problem as we saw with *PS-algol* (section 2.4.5 on page 21), namely that having set up a virtual page to point to its associated virtual block in a particular experiment it becomes necessary to undo the mapping whenever a new experiment is selected in order that the wrong virtual file is not used. The process of selecting a new experiment will therefore have quite a high associated overhead. The algorithm used to manage the system requires to do the following:

- When a new experiment is selected, the paths up the tree are compared to find the common parent (the root part is a parent to all experiments). All those pages which are defined in the path from the common parent to the old experiment are unmapped and replaced by the corresponding pages defined in the path from the root static part to the new experiment, setting the protection according as they are in the leaf experiment or its parent. Note that when performing the remapping we cannot just stop at the common parent node, since although any pages not redefined in the path from it to the old experiment must still be valid for the new experiment, those pages which the path redefined may be defined for the new experiment at any point on its path to the root. Any pages defined by the new experiment which have not been mapped by this process must also be mapped: these will be pages which have been defined in that part of the path not in common which were not defined in the non-common path to the old experiment. Again we must set the protection appropriately.

Figure 7-7: Example design tree



Consider how the algorithm would operate on figure 7-7 if we were suspending the experiment with *F'* and selecting that with *K*:

1. Find the first common parent experiment. This would be the one containing *F* and *G*.

2. Unmap and replace those pages defined by the old experiment. We unmap *\C*, *F'* and *J* and replace them with *C*, defined in the root, *F*, defined in the node containing *F* and *G*, and *J'*, defined in the new experiment.

3. Map *K*, defined in the new experiment but not yet mapped.

This algorithm, although being the most direct solution to the problem, has a number of disadvantages. In particular, the act of unmapping a page totally severs its connection with the operating system memory manager's tables: if the page has been written then it must be flushed out to disc, with the user being suspended until this has completed; and if the block is ever required again then it must be re-read from disc as it is no longer recapturable.

Furthermore, if we reselect the previous experiment without touching the newly mapped page then all the overheads of unmapping the old incarnation and mapping the new one will have been unnecessary.

By setting the protection on pages rather than unmapping them, and using an exception handler to field any resulting access violation faults we can avoid unnecessary unmapping and remapping of pages. The algorithm then becomes:

- As before, compare the paths up the tree to find the common parent. Set the protection on those pages defined in the path from the old experiment so as to make them inaccessible to *user* mode. If this is the first time the new experiment has been selected then set the protection on the pages defined on the path to the common parent so as to make them inaccessible to *user* mode; this guarantees that the pages will be correctly mapped by the access violation fault handler if they are ever used. Do not, at this stage, unmap any old pages or remap new ones.

- On an access violation fault to one of the protected pages, first check to see whether the page was mapped by a previous invocation of the experiment. If it was then it does not need remapped, and the resulting overheads can be avoided. If not, then unmap it and use the translation tables on the part from the current experiment to the root to remap the correct new page. Set the protection on the page to read-only or read/write according as it is in the leaf experiment or its parent.

- As with the previous algorithm, any access violation fault to read-only pages indicate that the page is being modified for the first time, and that a new site in the file requires to be allocated.

Again with reference to figure 7-7 we can follow the operation of the algorithm:

1. As with the previous algorithm, we first find the common parent (the node containing *F* and *G*).

2. Set the protection on those pages containing \C, *F'* and *J* to no access to *user* mode.

3. If this is the first time that the new experiment has been opened then set the protection on those pages containing *J'* and *K* to no access to *user* mode.

If the user attempts to access any of pages *A*, *B* or *G* then either a read operation will take place with no access violation fault occurring, or an access violation fault caused by attempting to write to a read-only page will occur, indicating that a first-modify is taking place, in which case the handler allocates a new site in the file. If an access is attempted to one of the protected pages, *F* say, then the handler will unmap the page defined by the old experiment (*F'*) and map the page defined by the new experiment (*F*).

If the old experiment is reselected then the protection algorithm is followed with respect to pages *J'* and *K*. The page containing \C, having not been touched by the other, experiment will still be protected, but if any attempt to access it is made then the exception handler will find that it is still mapped and will merely reset the protection (to *user* read/write), no remapping being necessary. The page may, of course, have to be re-read from disc if the system is busy, but it may be the case that retrying the instruction which caused the access violation fault will result in the page being recaptured with a consequential saving in disc I/O traffic.

Both the above algorithms require that the paths from the old and new experiments to the root be compared and the mapping of pages into virtual memory adjusted accordingly. The comparison is only really required for those pages which the new experiment is about to touch, however. This leads us to yet another possible approach, *viz* setting the protection on the entire database to allow no access at all to *user*-mode, with the access violation handler performing the path comparison only for those pages which are required for the new experiment.

All the above algorithms for selecting a new experiment require that the paths from the old and new experiments to the root be compared. At the expense of maintaining a table indicating where a page is defined, we can avoid scanning the old path, since the information as to where a page is defined is available for free as a by-product of mapping the page. Each time a page from a new experiment is required, the route from the leaf back to the root must be scanned. If, at that time, we record the position in the tree where the definition of the page was found, we can obviate the necessity for scanning the old tree when we map a new page. The algorithm for mapping a new page then becomes:

1. Scan down the tree, looking for the definition of the new page;
2. Compare the tree node where the definition is found against the node where the page was defined in the old experiment;
3. If the two are the same, then the page is common, and no more requires to be done; otherwise, unmap the old page, map the new page, and record the node where the new page was defined.

A two-byte quantity should be able to uniquely identify all the nodes on the experiment tree. In this case, one 512-byte page could hold definition information on 256 database pages, a space overhead of rather less than 0.5%.

Yet a further optimisation may be introduced by tagging the page with another two-byte quantity, indicating which experiment established the mapping. This will obviate the necessity to search the tree in those cases where a newly-reselected experiment discovers that no other intervening experiment has redefined the mapping for a page. Although this will result in a lesser saving, the space overhead is again less than 0.5%, and so this optimisation is probably a worthwhile one.

If the database is sufficiently small that mapping it does not use up a large fraction of the process's virtual address space quota then there is another possible algorithm: map each experiment into its own area of virtual memory. With this algorithm, selecting a new experiment costs almost nothing, as all the user (program) has to do is use a different base address when accessing the database. Sharing of pages and recapturing will be performed by the operating system. The main disadvantages are mostly determined by the size of the database: if it is large then the process may rapidly reach the limit of its virtual address space; and there will be no sharing of the cost of mapping a large database among a number of experiments, each of which may have only a small differential part compared to the size of the common fixed part. The former determines whether the method is practicable, while the latter affects the balance of overheads between this algorithm and the previous one. If the fixed part is large compared to the differential parts then the previous algorithm will tend to have lower overheads due to the sharing of setting-up costs described above; while if the differential parts are large compared to the fixed part, or switching between experiments occurs frequently, then the previous algorithm will have increased costs relative to the present one. Once again there is no universally best method, the characteristic access patterns of the application determining the relative costs of each approach.

7.5 Operating system requirements

In this chapter we have discussed the impact of virtual memory systems on virtual files with particular reference to *VAX/VMS*. We now generalise our discussion, asking what facilities are required from a host operating system in order that similar implementations may be made. The minimal requirement in order to implement experiments via the memory management system is the following:

- the ability to map an arbitrary block from a file into an arbitrary page in a process's address space;
- the ability to detect when a page was first written to, at which time it is remapped;
- the ability to flush pages out to disc and wait until they have been written; and
- the ability to remap a page while still maintaining its contents.

Under *VAX/VMS* we achieved the first requirement by accessing a (group) global section via an ASCII name,[1] with all the concomitant overheads, and the second by setting the protection on individual pages and fielding the resulting access violation faults. Note that for the latter scheme to work the memory protection must be on a per-page basis: the smallest structure which the memory protection mechanism believes in is also the smallest granularity at which we can usefully employ an exception handler to field access violation faults. The alternative requirement is that the operating system explicitly calls a user-supplied action routine the first time it finds that the hardware has set the "dirty" bit for a page; the action routine would be required to inform the memory manager of the new translation which is to be used for the

[1]Generated by the virtual file system, incorporating the process index and a sequence number to ensure uniqueness.

current, and any subsequent, transfers for that page. The last requirement we achieved by saving the data on the stack before unmapping the page, and restoring it when the new page had been mapped; clearly, however, this is not a particularly efficient approach to take. Since all that is really required is for the operating system's page tables to be updated to reflect the new site to which the page should be written, it should be quite practicable for such a facility to be provided.

In addition we found that the following features, while not essential, would be useful:

- the ability to indicate that while a particular page translation is not valid for the current experiment it should, nevertheless, be retained in case an experiment for which it is valid is reselected; or alternatively the ability to provide several alternative translations for a page in a manner analogous to our suggested persistent heap algorithm (section 2.4.5); and

- the ability to run the virtual file manager at a more privileged access level than "normal" user programs in order that its data structures may be protected against accidental (and malicious) damage.

Under *VAX/VMS* we achieve the first of these by explicitly setting the protection on individual pages. Again, for this scheme to work, the memory protection mechanism must operate on a per-page basis. The second may be achieved using the change-mode dispatching mechanism.

Chapter Eight
Design as a Co-operative Process

8.1 Introduction

In the previous chapters we have been concerned only with the case of a single designer working on a single aspect of a design, where there has been a one-to-one correspondence between design versions and experiments. We now turn to look at the case where a number of different aspects of a design are being worked on concurrently.

Consider the case where we have several teams of designers each working on a separate area of a design. Suppose one team is making experimental modifications to the database in the course of trying out a new design. It would be undesirable if these experimental changes became visible immediately to all other teams, since the "noise" which this would generate would interfere with the designs in adjacent areas. When the team nears the end of the design process, however, it becomes desirable that their proposed changes in the design should become visible to the other users of the database in a controlled way so that they may modify their own designs appropriately, with areas of conflict in the design being identified and some rectifying action taken.

Formerly we could regard an experiment as a complete unit of change, the interior workings of which were not visible to the outside world; in this sense it was akin to a transaction. This is no longer the case when we require that co-operating teams should be able to choose to incorporate the effects of part-completed experiments into their own designs, since the process of change has not finished by the time other users can see some of its effects. Neither can we regard an experiment as a transition from one consistent state to another, since although this may be true for some users it need not be true for all users; indeed some users may see the change as making the database inconsistent, because they are taking into account some other changes which the originator of the former is not.

This has the effect of transforming the design tree into a more general directed graph. The requirement that updates take place at the most "leafward" node implies that no awkward cycles can develop in it, however, since voluntarily incorporating another experiment can only be done by means of an update operation. This action is forbidden to nodes with any offspring at all, even adopted ones.

We make the following definitions:

- As before, an *experiment* is a related set of changes made by a designer to a database. They immediately become effective for that designer, though other designers will not see them unless they explicitly ask that the experiment be used in their own design.

- When a designer is satisfied with his experimental design he makes a *proposal* that it be incorporated into the definitive design.

- After being *reviewed*, the proposal will either be *accepted*, in which case it is incorporated as part of the main database and must therefore be taken account of by all other users, or be *rejected*, in which case the designer may *revise* it and make a new proposal.[1]

- A designer may, of course, choose to *abandon* his proposal at any time until it has been accepted.

The decision as to whether or not to accept a proposal is, essentially, an administrative one. The proposed design will undergo the same review procedures as would a design carried out in a more traditional manner, and will be subject to the same approval criteria. Having the changes isolated in an experiment brings added benefits here, since we may use the knowledge this gives us about the proposal to aid the evaluation of how it is likely to impinge on the overall design and on the other teams working on it. Acceptance of a proposal implies that all subsequently opened experiments should take the newly accepted proposal as their base design; however experiments which were in progress prior to the acceptance do not necessarily become immediately invalidated, since the merge of the newly accepted proposal into the base design would normally be postponed until such time as no other design team required to use the previous base design.[2] Those for which no conflict is discovered during validation (see section 8.3 on page 103) can use the new design immediately, while those for which a conflict is discovered will require to make appropriate changes.

Since experiments may have sub-experiments defined on them, a similar review will be required whenever it is intended to incorporate a sub-experiment into its parent. In this case, of course, the review process will be on a smaller scale, examination being required only against sub-experiments defined on the current experiment, though the same algorithms as for the base case may be applied.

One factor which may complicate the merge process slightly is the fact that previous

[1]Note the contrast with abandoning a transaction, where all the changes made in it are lost.

[2]There are clearly management issues here. These are outwith the scope of this thesis, however.

versions of a design may require to be maintained [Warman 79]. Although we could achieve this by never performing the merge operation, but instead requiring that all accesses should go via the newly accepted proposal's differential file, this would not be acceptable from an efficiency point of view. We should provide most efficient access for those uses of the database which are most common, rather than those which were current in the past but are now mostly of historical interest.

In order that we can maintain the previous views of the database we must ensure that, for each of them, the database appears precisely as it did when they were current. We can do this if, before we merge, we construct an "inverse" differential file which, when used with the new state of the database, gives the appearance of its former state.[1] As new proposals are accepted, a chain of former states appears, each of which takes its validity from its most recent successor state. If ever some former design state must be attained, all that is required is that the database be viewed via the layers of inverse differential files which have accumulated over the years. Access efficiency may not be particularly high, and indeed the views may have to be reloaded from some archiving medium such as magnetic tape, but as they will not be required often this is not of great consequence.

8.2 Optimistic methods of concurrency control

As part of the process of considering a proposal it would be desirable if we could determine how it would impinge on the other experiments open on the database. We can make use of some of the techniques of *optimistic concurrency control* [Kung 81, Menasce 82] to determine how the various experiments interact.

The essential difference between optimistic methods and *pessimistic* methods, such as locking, is that with the former it is assumed that conflicts between transactions will occur only infrequently, while with the latter it is assumed that conflicts will occur frequently. In the former case transactions are allowed to proceed without hindrance until they wish to commit, at which time conflicts are resolved by force-abandoning one of the offending transactions, with a consequent waste of processing time, while in the latter case effort is expended to make sure that transactions cannot perform incompatible operations on the database by requiring that they follow a locking protocol. In the example given in [Kung 81], that of two transactions performing concurrent updates on a B-tree of order 199 and depth 3 with 10^4 leaf nodes, the probability of conflict was found to be less than 0.0007; in such a situation following a locking protocol would be an unnecessary overhead in most cases.

Using the optimistic methods of [Kung 81] a transaction follows three phases: a *read* phase when the main database is read-only, any writes taking place to local copies; a *validation* phase where the changes made by the transaction are checked to see whether they would cause loss of integrity in the database; and a final *write* phase when, if the validation phase has succeeded, the local copies are made global. The concurrency control mechanism

[1]For safety, this should not be performed as part of the merge pass.

maintains four sets on behalf of the transaction: a *create* set, which records which objects have been created by the transaction; a *write* set, recording objects (locally) modified by the transaction; a *read* set, recording objects read by the transaction; and a *delete* set, which records those objects which will be deleted by the transaction should it eventually be successfully validated.

Serialisability (see definition (2.1) on page 12) may be guaranteed by the following scheme: assign to each transaction a unique transaction number; then for each transaction T_j with transaction number t_j, and for all T_i with $t_i < t_j$ one of the following must hold:

1. T_i completes its write phase before T_j starts its read phase; or

2. The write set of T_i does not intersect with the read set of T_j, and T_i completes its write phase before T_j starts its write phase; or

3. The write set of T_i does not intersect the read set or the write set of T_j and T_i completes its read phase before T_j completes its read phase.

The first of these conditions says that T_i completes before T_j actually starts; the second says that the writes of T_i do not affect the read phase of T_j; while the third says that T_i does not affect the read phase or the write phase of T_j. If none of these conditions holds then T_j is force-abandoned and restarted.

Since this approach is optimistic, transactions are assigned their transaction numbers at the start of their validation phase, thus allowing validation to proceed immediately. This may result in a long-running transaction being repeatedly failed, however. If this is found to be the case, then such a starved transaction may be restarted while still holding the semaphore which permits the critical section of validation to complete, in effect locking the entire database.

8.3 Optimistic methods and proposals

It will be seen that the validation phase of optimistic concurrency control is very similar in intent to that part of the review of a new proposal which attempts to discover how it impinges on other designers. The difference is that with the optimistic concurrency control methods described in section 8.2 the transaction which is validating is compared against those which have previously committed in order to determine whether there are any conflicts, the validating transaction being abandoned and restarted if this is found to be the case; with proposal review, however, it may be assumed *a priori* that the design is consistent with the current state of the database, since the designer would have taken into account all the revisions made to the base design before the proposal was made. In the latter case what is required is not to determine whether or not the proposal is consistent with the base design, but rather to determine which (if any) of the other concurrently-active designs will be affected if the proposal is accepted.

The requirement, then, is that the proposal being reviewed should be compared against the other concurrently active experiments in order to determine which, if any, of them will be affected if the proposal is accepted. We can achieve this by a variation on the optimistic approach of [Kung 81]. As before, we maintain read and write sets for each of the active experiments.[1] If an experiment becomes a proposal we compare its read and write sets against those of all the other currently-active experiments. Clearly, there is no problem if the read sets intersect; action will be required, however, if the read set of the proposal intersects with the write set of any of the other experiments or if the write set of the proposal intersects with either the read set or write set of any of the other active experiments. More formally, let R^* and W^* be the read and write sets respectively of the proposal. Let $\mathbb{E}$ be the class of active experiments, with R_i $(i \in \mathbb{E})$ and W_i $(i \in \mathbb{E})$ their read and write sets respectively. Then the proposal does not conflict with any active experiment if

$$\forall\ i \in \mathbb{E} \quad R^* \cap W_i = \phi \quad \text{and} \quad W^* \cap W_i = \phi \quad \text{and} \quad W^* \cap R_i = \phi .$$

In fact, since an experiment will have read an object before it writes it, we have that

$$W^* \subseteq R^*,$$

and hence the second of the conditions is implied by the first. This means that we do not require to compare write sets, since this will be done as part of the comparison of read against write sets.

This approach will indicate to the reviewing body the way in which the proposal conflicts with the other active experiments; it is then up to the reviewing body to decide whether the conflicting experiments are important enough that the proposal must be revised, or whether the proposal should be accepted in its current form, with the implication that conflicting experiments will require to be modified to take account of it.

The extent to which modifications are required depends on the degree of partitioning existing between the various design teams. If the areas of responsibility of the teams were kept (almost) disjoint when the design project was initiated then the areas of overlap will be correspondingly small; while if partitioning has not been effectively carried out then the possibilities of conflict are magnified accordingly. In any case there seems no possibility that resolution of such conflicts could be automated, since the decision necessarily requires knowledge and judgement (and political ability) beyond the computer's capacity; if this were not so, then the computer would have been able to carry out the design process itself.

8.4 An alternative optimistic strategy

Although not directly relevant to this thesis, it is interesting to note that the approach we are suggesting in section 8.3 may be used as the basis for an alternative strategy for optimistic concurrency control methods. Instead of informing the review body, the transaction manager allows any transaction which so requests to commit, at that time force-abandoning

[1]We subsume the create and delete sets into the write set.

and restarting any transactions which conflict with it. This scheme maintains database consistency by terminating transactions immediately it is discovered that they are not compatible with the current contents of the database. It is possible to allow a transaction to commit without requiring that it validate itself, since any transaction which would have been in conflict would have been abandoned before it declared itself to be committing.

In order to increase concurrency, validation of currently-executing transactions takes place in two phases: when a transaction commits, executing transactions must check their own reads and writes (not sets) against its (now static) read and write sets; at the same time the committing transaction checks the read and write sets of the active transactions. If the executing transaction discovers that it is in conflict with the currently-validating transaction then it abandons itself, while if the validating transaction discovers any executing transactions which conflict it force-abandons them. This obviates any necessity to hold up transactions while validation is under way, at the same time avoiding any requirement for interlocking of the accesses to active read and write sets. If the validation fails, the executing transaction is abandoned; if it succeeds, the executing transaction must use the newly-committed transaction's view of the database for all subsequent accesses. A final requirement that no transaction may commit until it has been validated against all transactions which have previously declared themselves to be committing maximises concurrency while preserving integrity.

While related to the algorithm of [Kung 81], this algorithm differs in the time at which it detects and handles transactions which are based on old or invalid data. In the former method, a committing transaction is checked against all those which have previously committed, while in the latter the currently active transactions are checked against the transaction which is committing. Each of these schemes has advantages and disadvantages: with the method of [Kung 81] a transaction may consume a large amount of system resources before it fails validation; while with the method proposed here, there may be unnecessary effort expended in checking a transaction which would, in any case, have voluntarily been abandoned at a later stage. The balance between the two methods is, again, application dependent: if transactions regularly consume large quantities of system resources or almost always commit, then the method proposed here would gain relative to [Kung 81]; while if transactions were small or were frequently abandoned, the converse would be the case.

8.5 Implementation

In order to support the review process described in section 8.3 we must maintain read and write sets for each experiment. Using differential files we find that the address translation table already contains all the information required for the write set, so that no further work is needed. Using shadowing we can derive the write set by comparing the LP-maps of the experiment and its parent, but the required information can not be obtained directly. In both cases we must arrange to maintain the read set, since this can not be deduced from the information in the maps.

Since the only information required for the read set is whether or not a given block is a member of it, we have a choice as to how we implement it. On the one hand we could use a similar technique to that of the translation table (write set), *viz* to record the addresses of each of the blocks in the read set in a hash-table or a B-tree or some such structure. We could also use a Bloom filter, as with the differential file system, in order to reduce the overheads of comparing read and write sets. Alternatively, since the information is essentially binary, we can use an "ordinary" bitmap. This has the advantage of faster access, since indexing into the table will avoid the need for a large number of comparisons. At the same time it allows a denser representation, assuming that accesses to the database are reasonably clustered, since a 512-byte block holds 4096 bitmap entries but at most 64 hash-table entries (ignoring any overflow chaining which would be required), with the result that caching in main store becomes practicable. If the database is large, with experiments generally not accessing more than a small fraction of it, then a simple bitmap will contain large empty areas. We can overcome this problem by using a two-level bitmap, akin to the two-level LP-map described in section 3.5 (page 34). In this scheme the top-level map would consist of pointers to the next-level bitmap, with 64 such pointers to a block. Thus a single top-level block could be used to maintain a read set over a 262144-block database, about half the formatted capacity of a 300Mb disc drive such as the *DEC RM05* or the *CDC 9766*.

Comparing the read and write sets requires that one of them be scanned and the other probed. Clearly, it would be most efficient if we scanned the smaller and probed the larger. Since the bitmap maintaining the read set will in principal span the entire database, although in practice a two-level scheme would reduce its size somewhat, while the write set (differential file translation table) will contain only those blocks which have been written to, it would seem most sensible to scan the write set and probe the read set.

In order that an executing experiment be succesfully validated, its parent must have previously been validated, since if the parent conflicts with the proposal then all its offspring must too. However, there is no need to validate offspring of other than the parent of the proposal, since the hierarchical nature of the design graph means that any such validation necessary will take place when the parent becomes a proposal at its own level.[1] This is in accord with the philosophy of experiments, which seeks to isolate designers from the noise of other users' trial designs. The algorithm used to review a proposal then becomes as follows:

1. Find the parent node.

2. For each of the parent node's offspring, both "natural" (genuine sub-experiments of the parent) and "adopted" (those which have elected to use the parent), validate the read and write sets against the proposal, noting conflicting values for further consideration by the review body; then call step 2 recursively to validate the sub-experiments' offspring.

It is the task of higher-level software, with a knowledge of the data structures and design algorithms, to interpret the conflicts in a way meaningful to the human reviewer. The virtual

[1]If the parent is the base level then the entire graph will, of course, be validated.

file manager has no knowledge of the contents of the blocks which it detects to be inconsistently defined between the two versions, and so can do no more than inform the higher-level system that such an inconsistency exists. The higher-level system, with its knowledge of the algorithms, data structures and schemata used, is much more able to interpret the differences and present them to the designer in a meaningful way. Interpretation may require knowledge of the context in which the changes took place. It might even be the case that the changes are consistent in some contexts but not in others, as for example in a common sub-assembly which is used in several disparate parts of the overall design. We believe that this is an important area of research, a solution to which will greatly aid designers' convenience and productivity. Having laid the foundations, however, we believe that it is beyond our remit to venture further.

Chapter Nine
Summary and Conclusions

In this thesis we have been concerned with the efficient implementation of systems to support computer aided design by experimental update. We believe that a designer using a CAD system should be able to attack his design problem in the most natural way, rather than being constrained to follow a particular methodology for no good reason except that the technology employed in the system he is using is insufficiently flexible to fully meet his requirements. It has been observed (for example [Eastman 78, Bobrow 80, Eastman 80, Goldstein 80a, Sidle 80]) that designers proceed by "trial and error", tentatively choosing a possible approach, developing it, evaluating it, perhaps discarding it for a different approach, developing and evaluating that one, eventually comparing all the possibilities and choosing the one which best meets the requirements. It seems only reasonable, then, that a CAD system should allow him such flexibility.

We call such a tentative collection of updates to the database an *experiment*. Experiments are initiated by being *opened* and terminated by being either *committed* or *abandoned*. At any time the designer may decide to *suspend* the current experiment and *select* another (thereby *resuming* it). Experiments may have *sub-experiments* defined on them, with these having *sub-sub-experiments*, and so on, the resulting *design tree* modelling the designer's tentative steps towards a final design.

9.1 Experiments and commercial database technology

Any attempt to meet our requirements using database systems designed for commercial data processing applications will, however, run into problems, as such an experimental approach to the database is entirely alien to their philosophy. In a typical commercial application, such as a banking system, a stock control system or an accounting system, the precise nature of the query or update is well defined in advance, since the nature of the world which is modelled in the database system is closely related to the nature of the business which is carried on by the organisation concerned. There is no concept in the banking world, for example, of provisionally cashing someone's cheque "just to see how it turns out", with the proviso that if the outcome is not satisfactory the transaction can be rescinded and some other solution tried. This, however, is precisely the sort of operation which we would like our CAD database system to support: if, for example, we substitute a VLSI chip for a collection of random MSI and LSI logic, how do the two variants of the overall design compare?

Another facet of commercial applications is that there may be a number of concurrent transactions accessing the database simultaneously. If the database is to be maintained in a consistent state these concurrent accesses must be carefully controlled (section 2.2.1 on page 11). This is traditionally achieved by means of a *locking* protocol (section 2.2.2 on page 13), whereby a transaction informs the database system as to the nature of the operations it wishes to perform and the data which it intends accessing. The database manager checks these requests against the requests it has received from the other concurrently-executing transactions to determine if the requested access modes are compatible or not. If the access modes are compatible then the transaction is allowed to proceed; if they are not compatible then the transaction is suspended until such time as some other transaction releases its conflicting locks. Depending on the locking protocol in use, there is the possibility of deadlock, with two or more transactions each waiting for the other to release its locks before they can proceed. This situation may be prevented, at the expense of a reduction in concurrency, by following a deadlock-free protocol; alternatively, the lock manager may detect the deadlock and resolve it by forcing one or more of the offending transactions to release its locks and restart. To be effective, such a locking protocol must require a transaction to acquire appropriate locks before it is allowed to access any item of data; the scheme can not work if there are some items of data, some access modes or some transactions which are exempt.

A locking scheme, however, makes the implementation of experiments completely impossible. The very nature of experiments implies that there will be two or more concurrent accesses to the same part of the database, as each experiment embodies a particular set of tentative steps towards the final solution of the current problem. A locking protocol requires that an experiment must acquire shared locks on all of the data in the database which it is going to read, and exclusive locks on those data which it is going to modify. These locks cannot be released until such time as the experiment either commits or is abandoned. However the very fact that the experiment has exclusive locks to part of the database immediately implies that no other experiment is allowed to acquire any locks at all on that part of the database. This is a fatal flaw in its ability to support a different tentative design version. Hence a locking protocol is incompatible with experiments.

9.2 Implementation techniques

One of the essential requirements of the sort of system we are proposing is that each experiment should appear to have its own private copy of the database to modify as it desires, totally independently of the modifications made by any other experiment, with the exception that since any sub-experiments will be relying on their parent experiment's view of the database remaining unchanging, since their own views are defined with respect to that of their parents, any experiment with offspring becomes read-only until such time as all its offspring either commit or are abandoned. The techniques summarised below were introduced in chapter 3 (page 24).

9.2.1 Copying

The most obvious way of making it appear that each experiment has its own private copy of the database is to give each its own actual physical copy. This, however, incurs severe space and time penalties: each time an experiment is opened the entire database must be copied; furthermore, the fact that there are a number of experiments active at any particular time implies that there will be a corresponding number of copies of the database in existence. If the database is at all large this may cause severe problems, with disc quotas and physical disc capacity being the ultimate limiting factors. Committing and abandoning are quite straightforward, however, the former being done merely by deeming the committing experiment's copy to be the definitive copy, with the latter being achieved by deleting the entire abandoning experiment's copy of the database.

9.2.2 Repeated editing

By holding the instructions which caused the updates made by an experiment rather than the results of such instructions it may be possible to achieve a very compact representation [Zhintelis 79]. For example, the instruction "use 5% tolerance resistors rather than 10% tolerance" may be stored in a much more compact way than the actual results of such a change, with the saving increasing with number of objects modified. This saving in space is achieved at the cost of an increase in the time cost of resuming an experiment, however, which may make the use of this method prohibitively expensive. It is possible to ameliorate the time cost at the expense of space by holding several working copies of the database in "expanded" form; this runs into the problems described above, however. An experiment may be abandoned by deleting the instructions which obtain it from the base design, committing taking place by applying the update instructions to the base design.

9.2.3 Differential files

If that part of the database which has not been altered by any experiments could be shared, then we would expect a great saving in space overheads to result, particularly with large databases, since in such a case an experiment would not be expected to modify more than a small fraction. The method of *differential files* [Severance 76] divides the database into two parts: a read-only static part, and a writeable differential part which contains all the modifications made to the database. Here we share the static part among all the current experiments, thereby achieving our goal of space-efficiency, and give each its own differential part, thereby isolating experiments from each other while allowing them to change the database at will. Accesses to any item in the database are first checked to see if they are in the differential part, with the value there being used in preference to that in the static part. Note that by using a parent experiment's virtual file as the static part and creating a new differential file we have a convenient implementation of sub-experiments.

Checking the differential part for the presence or absence of an item may be speeded up by using a *Bloom filter* [Bloom 70]. This technique uses a bit-table and one or more hashing functions to give an indication as to whether an object is in the static part or the differential part. If all the bits accessed via the hashing functions are set, then the object may be in the differential part; while if any of the bits is unset, then the object is definitely not in the differential part. In those cases where the filter indicates that an object may be in the differential part, the main translation table must be searched to provide either the address of the object in the differential file or a positive indication that the object is not in the differential part. Thus, although the filter does not give a definite decision as to whether an object is in the static part or the differential part, in those cases where there is some doubt it errs on the side of safety by indicating that the differential part's directory should be checked to provide a definitive answer. Accessing the filter will be a quick operation, since the entire bit-table may be kept resident in main memory, whereas the differential part's translation table will, in general, be too large for it to be practicable to cache it in main store, and hence one or more slow disc accesses may be necessary. Thus, although there is a CPU cost in using a filter, in general it will be more than compensated for by the reduction in disc traffic, giving an overall saving.

Abandoning an experiment is straightforward: the differential file is merely deleted. From an efficiency point of view, committing requires that the differential part be merged into the static part, though there is no reason to restrict access to the database while the merge is taking place since, if all accesses go via the committing differential file until such time as the merge has completed, the database has the same logical appearance whether merged or not. For the same reason, the integrity of the database is quite safe while the merge is taking place; if the process is interrupted, by a system failure, for example, the merge can be restarted from the beginning.

9.2.4 Shadowing

Shadowing [Astrahan 76, Challis 81, Gray 81b, Traiger 82] is another technique which allows the sharing of common parts of a database. In this approach, all the blocks in the database are accessed via a *logical-to-physical* map (LP-map). When an experiment is opened it takes a copy of its parent's LP-map. Thereafter, each time it modifies a block for the first time, a new site in the file[1] is allocated and the LP-map adjusted accordingly. An experiment commits by having its LP-map used as the definitive map in place of that of its parent; abandoning is done by deleting the LP-map. In the case both of committing and abandoning, the database must be garbage-collected, though the user need not be kept waiting while this is done. Garbage-collecting may be a lengthy process, since, as we saw in section 3.5, the LP-map which is being deleted must be compared against every other LP-map, in order to determine which blocks are still in use, although the use of a two-level LP-map will help speed the process since the equality of two top-level entries guarantees the equality of all the corresponding sub-entries.

Again we can implement sub-sub-experiments easily, this time by ensuring that the parent sub-experiment's map is copied, rather than that of the root design.

[1]or disc, if there is no host filing system in use

9.2.5 Comparison of differential files and shadowing

Differential files and shadowing are clearly closely related; indeed, some would even consider them to be the same. There is, however, an important logical distinction between the two approaches: with shadowing, both the changed and unchanged blocks are held as part of the same structure; the differential files approach, however, maintains a clear distinction between the unchanging static part and the changing differential part. This has a number of important outcomes: using shadowing, the same addressing mechanism must be used for both the changing and unchanging part, with the result that, although it may be optimal for one of them, it need not be optimal for both; different addressing mechanisms may, however, be used for the static part and the differential part using the differential files approach, with the result that they may be separately optimised. In particular, at the lowest level an extent-based addressing scheme may be used, with a consequent reduction in the space and time overheads required to maintain the database as compared with the shadowing approach. Garbage-collection is greatly simplified for differential files, since all the related changes are grouped together, and is, in any case, merely the normal operation of deleting a file. The database may be better protected against corruption by write-locking the static part, either physically or logically. Finally, system backup is facilitated, since the static part will only require to be backed up once, with the differential parts being accounted for in the regular incremental backups of the host filing system; this important operational advantage is denied to the shadowing approach, since, to the host filing system, the entire database appears to have changed and so requires to be backed up.[1]

Note that all the internal mechanism of implementing experiments by such means is hidden from the user: all that appears at the top level is a structure which has all the appearance of an "ordinary" file, albeit with a number of additional properties. We call this structure a *virtual file*, since it gives the user the appearance of corporate existence while in reality being synthesised by the underlying system.[2]

9.3 Differential files evaluated

Of the techniques described above, we believe that that of differential files provides the most likely basis for a sound and efficient implementation of experiments. There is little point in such a system, however, if it is so slow and inefficient to use in practice that it impedes the designer more than it aids him. Both [Eastman 80] and [Sidle 80], for example, consider that speed is one of the major deficiencies of commercial database systems in CAD applications. In order to show that our suggestions are practically feasible as well as theoretically interesting it was necessary to build a system and measure its performance. In fact, two systems were built and tested, as described in chapters 4 to 6.

[1]The copying approach also suffers from this handicap.

[2]cf *virtual memory*

9.3.1 The Rbase implementation

In order to gain some experience in implementing a system based on differential files, a preliminary investigation based on the *Rbase* system [Tate 81] was undertaken (chapter 4 and appendix A). This system, originally written for *EMAS* by Austin Tate, provides the user with a data model which has features of both the relational and network approaches, and of Artificial Intelligence languages such as *CONNIVER* [Sussman 72]. The original system interfaces to the *EMAS* operating system by means of a number of store-map files, the contents of which are directly manipulated by the program rather than going through a central buffer manager. The preliminary investigation was performed before the *Shrines* technology (section 7.2) was developed, with the differential file being interposed between *Rbase* and *VAX/VMS* by substituting explicit maps for the *Imp77* built-in maps *integer*, *byteinteger* and *string*. Although this slowed the system down, it was nevertheless still useable. In order to reduce the translation tables to a manageable size an artificial blocking was imposed on the system; whenever any byte in a block was modified for the first time the entire surrounding cell required to be copied to its new site and the update applied there. The alternative to this would have been, potentially, a separate translation for each individual byte, an overhead of at least 800% (allowing for both the key address and its translation).

The performance of the system was measured by loading 250 sample tuples and posing a number of queries. The following were measured:

- CPU time
- Page faults
- Elapsed time

Four database system configurations were measured:

- No differential file
- Differential file
- Differential file with address cache
- Differential file with no filter

Use of the differential file, both with and without the address cache but with the filter, showed a two-fold increase in CPU time and a three-fold increase in elapsed time and page faults over the non-differential file case. The results for the test with the filter disabled, however, showed a substantial increase in CPU time, elapsed time and page faults over the non-differential file case, demonstrating that the filter is a necessity and not a luxury.

Copying surrounding data is an unavoidable overhead unless differentiation takes place at the individual byte level. Decreasing the size of the differentiation cell in an attempt to reduce the copying overhead inevitably results in an increase in the cost of translation due to the larger structures involved, which seems to result in an impasse. If we are prepared to accept the space overhead of taking the disc block as the unit of transfer, however, then the copying operation comes for free as part of reading from disc, provided that we write the modified block back to a new site.

Approaching the same problem from the other end, we note that it is common practice for database systems to group related objects in the same disc block for reasons of efficiency of access. It is necessary for us to preserve such structure in our virtual file system in order to provide adequate support for the end-user CAD system. We are, once again, driven to the disc block as the unit of differentiation.

9.3.2 The block-based implementation

Having determined that the differential files technique was sound in principle, a second implementation was undertaken (chapters 5 and 6 and appendix B). This was intended as a basis upon which applications could be built, not as a complete system *per se*. The interface to the applications program presented the appearance of a virtual "direct access" file, with procedures for opening and closing files and for reading and writing individual blocks, as would also be the case for "normal" direct access files, together with a procedure for merging a differential file with its corresponding static part. Such an approach, while possible under almost any operating system, is of most relevance to those machines without virtual memory support, since in this latter case additional techniques may be used to enhance performance and provide a more convenient user interface.

As the system was not targeted towards any particular application, the method of testing its performance required to be similarly general. It was decided that, rather than try to guess at "typical" access patterns, a random number generator (provided by Gordon Brebner) should be used to choose blocks for reading and writing. This provides a more severe test than would most "real" applications, since the complete lack of locality of access and the fact that blocks are touched once only and then discarded means that head movement is increased and the cache is rendered much less effective than it would be with a more regular access pattern. Hence, if the system performance was found to be satisfactory under this test regime then it should also be satisfactory for a real application. Tests were carried out, for the most part, on the Data Curator Project's *3220*, although one test on the Computer Science Department's *VAX-11/780* showed that the performance there was qualitatively the same. The *3220* was chosen partly because it could be used as a "single-user" machine, and partly because timing is carried out more reliably under *Mouses* than under *VMS* [DEC 81, Culloch 82].

The tests involved reading and writing a number of randomly-chosen blocks, and recording a number of measurements on I/O traffic, elapsed and CPU times and performance of the cache and the filter. Tests were performed both with and without differential file layers and for varying cache sizes, cache disciplines, translation hash-table sizes, depths of differential file layers and sizes of base static part. Both "large scale" tests and "small scale" tests were performed; the former involved reading and writing 5000 blocks to the base part and 1000 blocks to the static part, while the corresponding figures for the latter were 20 and 100.

The first test performed, both on *VAX* and the *3220*, was intended to show the basic viability of the differential files technique as implemented. It involving reading and writing blocks both directly to the base part with no differential file layer and via a differential file. The results for the large scale tests show that, while there is a modest increase in total elapsed time, as a consequence of increased CPU time and disc traffic, this is small when it is remembered that there is no processing of any data in the virtual file taking place at all. There should not, therefore, be any unacceptable increase in overheads using a virtual file as compared with accessing an "ordinary" disc file directly. For the small scale tests, the differences in the figures are so small that it is doubtful if a user would notice them, particularly on a time-sharing system.

As we saw with the *Rbase* implementation, the filter is necessary if adequate performance is to be obtained. The effect on the block-based implementation of disabling the filter was to increase total elapsed time by around 25%, mostly attributable to a 40% increase in disc reads, required to bring in translation blocks which have been flushed out of the cache to make way for more immediate requirements. As the filter size is only one block it would seem to be entirely false economy to attempt to dispense with it. This test was performed in the large scale version only, the results being so conclusive, and the cost of implementing a filter so slight, that there seemed little point in running the small scale version.

The choice of cache size is a trade-off between the saving in I/O costs achieved by increasing it and the extra overheads incurred thereby. There is no real point in keeping blocks in the cache which are no longer required, as this merely results in extra unnecessary comparisons. The size of the cache was varied, in steps of 8, from 8 blocks to 40 blocks, and the resulting performance variations measured. For the large scale test, the conclusion was that, for the random access pattern of employed, there was some gain to be had in increasing the cache size, but that this was less for larger cache sizes, the reason for this behaviour being that for larger caches the most frequently required translation blocks (those in the hash-table and near the front of the overflow chains) become resident, with any further increase merely resulting in data blocks taking longer to become "least recently used". In practice, of course, we would have to take account the number of data blocks expected to be simultaneously accessed, increasing the cache size accordingly. Again, the small scale test shows no evidence of noticeable variations in performance.

The virtual file system does not know anything about how the data blocks it is asked to handle are actually used; in particular, it has no way of knowing that a particular block is not going to be required again, and that its slot in the cache may be reused. If higher-level software is able to guide the cache manager as to whether particular blocks may be discarded then performance may be improved. This was tested by comparing the cache manager's "least recently used" strategy with a "toss immediate" strategy based on the test program indicating that the block it had just accessed was no longer required. The results for the large scale test showed a 12% decrease in elapsed time for the "toss immediate" discipline as compared with "least recently used", with a similar result for the small scale test. Allowing higher-level software to guide the cache manager's decisions as to when to release blocks can, therefore, result in substantial and worthwhile system performance enhancements.[1]

[1]Prefetching would be rather harder to implement.

Surprisingly, perhaps, increasing the translation hash-table size made very little difference to the measurements obtained in the large scale test, while performance was actually degraded in the small scale test. This can be accounted for by the action of the cache in reducing disc traffic for the blocks holding the overflow chains. A large table results in a greater number of shorter overflow chains, while a small table results in a lesser number of longer chains, with the frequency of access of a particular overflow block becoming proportionally less as the size of the table is increased. Moreover, if the table is too large, the blocks of the table itself may be accessed insufficiently frequently to prevent them being flushed out of the cache. All in all, it seems that a table size of around 4 to 8 blocks would result in an acceptable performance in most cases.

If a tree of experiments is to be supported adequately, system performance must not degrade significantly as extra differential file layers are added. The effectiveness of the filter, however, resulted in any differences which may have existed being submerged well below the noise threshold of the test which measured this effect. A slight increase in CPU time was noted as extra layers were added, but no other measurement could be seen to have changed.

In all the experiments described above, the size of the filter bit-table was such that false positives could not arise. While the space overhead of the filter is not great (less than 0.025%) so that it makes sense to ensure that it is large enough, it would be desirable that performance was not significantly degraded in those cases where the virtual file is of sufficient size that false positives may occur. Comparing performance for virtual files of size 16384 blocks against that for 4096 blocks for both a large and a small scale of update, it was found that total elapsed time increase in both cases by around 18%. When the same test was repeated using "ordinary" DA files, it was found that elapsed time increased by around 23%. We conclude that any increase in overheads engendered by the incidence of false positives has been swamped by the increase in disc latency due to seeks being required more frequently and over a larger distance.[1]

Overall, then, the performance of the virtual file manager based on differential files was quite acceptable, with the performance degradation resulting from using a differential file, as against accessing the base file directly, ranging from modest in the case of a large scale of updates to unnoticeably small in the case of a small scale of update. It would, therefore, be quite practicable to use a virtual file system based on differential files as a basis for CAD by experimental update.

We predicted that the much larger translation data structures of the shadowing approach would result in it performing rather less well than the differential files approach, and indeed this is confirmed in practice. Although requiring less CPU time than the differential files approach, shadowing generated 45% more disc traffic than did differential files, resulting in an increase in elapsed time of 27%. Given the operational advantages of differential files over shadowing, and the simplified implementation of experiments, we conclude that the method of differential files is to be preferred to shadowing.

[1]The operating system does not issue a seek if the heads are already on-cylinder as a result of a previous transfer.

9.4 Virtual memory

In view of the complications involved in using a buffer manager, as compared with the simplicity of access to disc files afforded by the virtual memory management facilities provided by such systems as *EMAS*, it is only natural to ask if some form of direct mapping can be used to improve an implementation of experiments. This would be expected to result in two benefits:

- there would be no need to search the cache on each access to the database, as this would be performed implicitly by the address translation hardware; with the result that

- the user would no longer be tempted to assume that a particular buffer contained a particular disc block, a rather dangerous assumption to make since the cache manager may have rolled out the block to make way for a new one.[1]

There are three possible approaches to database access on a virtual memory machine:

- *Small cache*, locked in memory so as to avoid paging. This was the scheme used in the block-based implementation described above.

- *Large cache*, paged along with the rest of the process's working set.

- *Directly mapped*, as, for example, *System-R* [Traiger 82], where that part of the database which is "of interest" is mapped, the system described by [Grimson 80], or the *Shrines* system (chapter 7 and appendix C).

9.4.1 Implementing transactions under VAX/VMS

As a first step towards a full implementation of experiments we considered the implementation of transactions. The *Shrines* system uses shadowing to achieve this effect. By making use of the memory protection mechanism it is possible to detect when a page is first written, with the mapping between disc file and page in virtual memory being adjusted at that point to take account of the new address allocated.

9.4.2 Overheads under VAX/VMS

In order to compare the performance to be expected from the schemes described above, a number of tests were performed involving reading and writing random pages from a section file on a user disc and the system page file, both by using explicit QIOs and implicitly

[1]Although it would be possible, with care, to remember that a particular buffer corresponded to a particular data block, it is fraught with even more danger than remembering the contents of machine registers in assembly-language programming.

transferring pages via the paging mechanism with both the default page fault cluster size and with a page fault cluster size of one, the latter most nearly simulating the internal fragmentation caused by the shadowing scheme used in the *Shrines* system. Plotting total elapsed time (in milliseconds) against the number of pages accessed showed the effects of a number of *VMS* paging mechanisms, most noticeably the effects of page recapture and of read clustering, and, to a lesser extent, of deferred writing of modified pages.

The results showed that under *VMS* the directly mapped scheme with a page fault cluster size of one (*Shrines*) would be expected to outperform the large cache scheme in those cases where there would be at most two distinct access periods to a page, with the lower overheads incurred by using the system page file compensating for the higher cost of explicit QIOs and the "unnecessary" paging which may take place when the blocks are written from the cache. Differential files, however, would not suffer the same degree of internal fragmentation, and so would be able to make use of clustered reads from the section file holding the static part. The benefit of this would be most marked if the access pattern showed more clustering, as would be expected from a "real" application, and in this case the mapped scheme would be clearly superior.

9.4.3 Experiments under VAX/VMS

Although it may be possible to obtain performance gains for a transaction system by exploiting virtual memory management, for our purposes it must also be possible to implement experiments. In fact, as we saw there are three possible ways of achieving this. They are:

1. When a new experiment is selected, compare the design tree as seen from the old and new experiments, unmapping pages which are no longer valid and replacing them with those which are defined by the new experiment.

2. As above, but instead of unmapping and remapping, set the protection on invalid pages to be no access; perform the unmapping and remapping if there is an access violation fault for that page. This approach has the advantage that the relationship between an address and its corresponding page is not severed until absolutely necessary, thereby allowing recapturing to take place should the mapping become valid again (by reselecting the old experiment, or possibly by selecting a new one).

3. Protect the entire database, only performing the tree comparison for those pages which are actually touched when the new experiment is used. This has the advantage of reducing unnecessary processing overheads, though at the cost of making the exception handler more complex.

The tree comparison may be facilitated, at the cost of an increased space requirement, by tagging each separate node on the design tree and saving the tags of the nodes which defined each of the pages in an array, indexed by page. Then, when it is required to check whether a page translation is valid or not, all that is required is that the path from the current leaf experiment to the root of the tree be traversed and the tag of the node which defined the current translation of the page in question be compared against that of the node which defines it in the new experiment. If they are the same, no further action is required; otherwise the page must be unmapped and remapped with the new translation.

9.5 Co-operating teams of designers

In all the foregoing we have been assuming that we have a single designer working on a complete design. Although we feel that this is an important case, and that the results we have obtained show that it is quite practicable to support experiments in such a case, nevertheless it would be desirable if we could extend our concepts to allow for teams of designers working concurrently on (almost) disjoint parts of a large design project.

Our requirements are now two-fold: designers must be isolated from the "noise" generated by other designer's tentative designs; and when a design reaches the stage of being finalised and incorporated into the main database it is necessary to discover which, if any, other designers will be affected by the changes.

We defined the following terminology: when a designer is satisfied that his design meets the required specification he makes a *proposal* that it be incorporated into the main design database; the proposal is *reviewed*, as would be the case were the designer working with current tools; the proposal is then either *accepted*, in which case all other users of the database must take account of it. At this time it will be merged with the main database, the previous state having been preserved in an inverse differential file in order that it may be accessible if required.

9.5.1 Optimistic concurrency control and proposals

Optimistic concurrency control methods differ from the more usual *pessimistic* methods such as locking by assuming that conflicts between transactions will be infrequent. If this is the case then it is more efficient to deal with any conflict after it happens rather than expending effort preventing it happening in the first place. This is very similar to the detection of conflicts between proposals and other current experiments, and so we would expect that similar techniques would be required. In fact, optimistic concurrency control as originally suggested [Kung 81] is inappropriate, as it is concerned with detecting conflicts between the committing transaction and those which have previously committed, rather than, as here, between a proposal and those experiments which may become proposals in the future. A variant on the technique (section 8.4) can, however, be used to achieve the desired result.

As with the original version, we require that each experiment maintain *read* and *write* sets, indicating which pages of the database it has touched. By comparing these read and write sets we may establish which, if any, experiments conflict with the proposal under consideration. If differential files are used, then we may derive the write set from its translation table. The read set must be maintained separately, however, the method we suggest being a two-level bitmap, similar in concept to the two-level translation table described for the method of shadowing (section 3.5).

It should be noted that the decision as to what action to take, should any be required, must be left to the review body, as the value judgements involved are beyond the ability of the computer. The most that can be achieved is to produce as much relevant information as possible, so that the review process will be soundly based.

9.6 Envoi

In this thesis we have shown that the implementation of a virtual file system which provides the required facilities to allow design by experimental update is both desirable in principle and feasible in practice. Traditional techniques, involving *in situ* updates and locking being unable to handle experiments adequately, a number of alternatives were considered. Our analyses suggested that the *differential files* approach was most promising, particularly when used with a *Bloom filter*, and an extensive series of tests confirmed that the performance of such a system was, indeed, quite sufficient to meet our requirements. Virtual files could be used both with conventional database systems using commercially-oriented data models, and with a persistent programming approach such as that of *PS-algol* and *NEPAL*.

There are a number of benefits to be obtained by using a system of the sort we are proposing here. From the point of view of the designers, the constraints on their method of working are greatly reduced. This will result in a more natural design environment, with more possibility of working with the system rather than fighting against it. Quality and quantity of designs can only increase.

From the management point of view, ready access to the states of the individual designs, and in particular how they differ from the base design and from each other, will enable better overall monitoring of the project. Areas of conflict can be identified sooner, and hence resolved more easily and cheaply. It will be possible to detect designers straying beyond their remit, either because of a lack of clear understanding of its boundaries or because they were wrongly defined in the first place, and take appropriate measures by redefining or clarifying the partitioning.

From the end-user's point of view, the finished product should be both cheaper, because the design effort has been more efficiently applied, and better, because alternatives can be explored at an early stage and problems detected while there is still time to correct them. Encouraging designers to try out different designs will mean that there is more chance of the best design emerging, rather than being stifled at birth by the constraints of the system being employed.

A number of topics remain to be researched. Work is beginning, under the auspices of the Data Curator project at the University of Edinburgh, to investigate directly-mapped persistent heaps, although it is too early, as yet, to say how this will turn out. It is anticipated, however, that the results may form the basis of another Thesis.

In this Thesis, while we showed how experiments and proposals relate to each other, and how the information maintained by the virtual file manager may be used to determine which experiments are in conflict with a proposal under review, we deliberately excluded from our remit that part of the system which would be required to interpret the conflicting areas in a way which would be meaningful to the reviewing body. We believe that this problem can not be solved in general, although it is quite possible that higher levels of software, with a knowledge of the data structures and schemata in use, would be successful in particular cases. Although we have laid the groundwork, we recognise that a great deal of work remains to be done in this area.

Amongst the problems to be solved include those of handling alternative designs, interpreting the differences between them, and presenting those differences in a way which is meaningful to the user. Most current programming languages and systems assume that there can be only one valid state attained by a particular object at any one time; in contrast, we require that a number of different possible states be simultaneously accessible for comparison purposes. Interpreting the differences requires a knowledge of the data structures and algorithms used in the particular application. It may be possible in some cases to use particular knowledge of the area of application in order to aid this process. Again, presenting the information to the designer depends on the particular area of application and on the facilities used by the designer: if he is used to interacting graphically with the system then he will expect that areas of conflict will be highlighted graphically, while if he is used to interacting textually he will expect a textual response. Even to begin to investigate some of the problems involved would require specialised knowledge of the area of application.

While all the virtual file systems described in this Thesis are prototypes, they were designed on the assumption that they would be useable in "real life". None of them, however, incorporates all of the ideas presented above, as they were designed to test particular aspects rather than to provide a complete system. It is intended to re-engineer them into a "production" version now that the technology has been proven, in order to provide a sound basis for further exploration of higher-level systems.

References

[Astrahan 76] Astrahan, M. M. *et al.*
System R: relational approach to database management.
ACM Transactions on Database Systems 1:97-137, June, 1976.

[Atkinson 78a] Atkinson, M. P.
Programming languages and databases.
Technical Report CSR-26-78, Computer Science Department, University of Edinburgh, 1978.
Also in *Proceedings of the Fourth International Conference on Very Large Databases* (1978).

[Atkinson 78b] Atkinson, M. P., Martin, J. and Jordan, M.
A uniform modular structure for databases and programs.
Technical Report CSR-33-78, Computer Science Department, University of Edinburgh, 1978.

[Atkinson 81a] Atkinson, M., Chisholm, K. and Cockshott, P.
PS-algol: an Algol with a persistent heap.
Technical Report CSR-94-81, Computer Science Department, University of Edinburgh, 1981.

[Atkinson 81b] Atkinson, M. P., Chisholm, K. and Cockshott, P.
The new Edinburgh persistent algorithmic language.
In M. P. Atkinson (editor), *Database*, pages 299-318. Pergamon Infotech, 1981.
Also available as Technical Report CSR-90-81, Computer Science Department, University of Edinburgh.

[Bhargava 81] Bhargava, B. and Lilien, L.
Feature analysis of selected database recovery techniques.
In *AFIPS Conference Proceedings*, volume 50, pages 543-554. American Federation of Information Processing Societies, 1981.

[Blasgen 77] Blasgen, M. W. *et al.*
System R: an architectural overview.
IBM Systems Journal 20:41-62, 1977.

[Bloom 70] Bloom, B. H.
Space/time trade-offs in hash coding with allowable errors.
Communications of the ACM 13:422-426, July, 1970.

[Bobrow 80] Bobrow, D. G. and Goldstein, I. P.
Representing design alternatives.
In *Proceedings of the Conference on Artificial Intelligence and the Simulation of Behaviour*. July, 1980.

[Brice 77] Brice, R. S. and Sherman, S. W.
An extension of the performance of a database manager in a virtual memory system.
ACM Transactions on Database Systems 2:196-207, June, 1977.

[Buchanan 82] Buchanan, I.
SCALE - a VLSI design language.
Technical Report CSR-117-82, Computer Science Department, University of Edinburgh, 1982.

[Buneman 82] Buneman, P., Frankel, R. E. and Nikhil, R.
An implementation technique for database query languages.
ACM Transactions on Database Systems 7:164-186, June, 1982.

[Challis 78] Challis, M. F.
Database consistency and integrity in a multi-user environment.
In B. Schneiderman (editor), *Databases: improving usability and responsiveness*, pages 245-270. Academic Press, 1978.

[Challis 81] Challis, M. F.
Version management - or how to implement transactions without a recovery log.
In M. P. Atkinson (editor), *Database*, pages 435-458. Pergamon Infotech, 1981.

[Chamberlin 76] Chamberlin, D. D.
Relational data-base management systems.
ACM Computing Surveys 8:43-66, March, 1976.

[CODASYL 78] CODASYL Data Description Language Committee.
Journal of Development.
Secretariat of the Canadian Government EDP Standards Committee, 1978.

[Codd 70] Codd, E. F.
A relational model of data for large shared data banks.
Communications of the ACM 13:377-387, June, 1970.

[Culloch 79] Culloch, A. D.
Mouses User Manual
Computer Centre, Moray House College of Education, Edinburgh, 1979.

[Culloch 82] Culloch, A. D., Robertson, P. S., Ross, G. D. M. and Whitfield, C.
Mouses V9.2 system sources.
1982.

[Date 79] Date, C. J.
Locking and recovery in a shared database system: an application programming tutorial.
In *Proceedings of the 5th International Conference on Very Large Databases*, pages 1-15. 1979.

[Date 81] Date, C. J.
The Systems Programming Series: An introduction to database systems, 3rd edition.
Addison-Wesley, Reading, Massachusetts, 1981.

[Date 83] Date, C. J.
The Systems Programming Series: An introduction to database systems: volume II.
Addison-Wesley, Reading, Massachusetts, 1983.

[DEC 77a] Digital Equipment Corporation.
VAX11 Software Handbook
1977.

[DEC 77b] Digital Equipment Corporation.
VAX11/780 Architecture Handbook
1977.

[DEC 78] Digital Equipment Corporation.
VAX11/780 Hardware Handbook
1978.

[DEC 81] Digital Equipment Corporation.
VAX/VMS internals and data structures
1981.
Order number AA-K785A-TE.

[DEC 82a] Digital Equipment Corporation.
VAX-11 Record management services reference manual
1982.
Order number AA-D031D-TE.

[DEC 82b] Digital Equipment Corporation.
RMS Utilities
1982.
Order number AA-M554A-TE.

[DEC 82c] Digital Equipment Corporation.
VAX/VMS System Services Reference Manual
1982.
Order number AA-D018C-TE.

[DEC 82d] Digital Equipment Corporation.
VAX/VMS real-time user's guide
1982.
Order number AA-H784B-TE.

[DEC 82e] Digital Equipment Corporation.
VAX/VMS system management and operations guide
1982.
Order number AA-M547A-TE, with update notice number 1, order number AD-M547A-T1.

[Denning 70] Denning, P. J.
Virtual memory.
ACM Computing Surveys 2:153-189, September, 1970.

[Dewar 83] Dewar, H. M., King, M. R. *et al.*
APM reference manual.
Technical Report, Computer Science Department, University of Edinburgh, 1983.

[Dodd 69] Dodd, G. D.
Elements of data management systems.
ACM Computing Surveys 1:117-133, June, 1969.

[Eastman 78] Eastman, C.
CAD: the next ten years.
Computer Aided Design 10:347-349, November, 1978.

[Eastman 80] Eastman, C. M.
System facilities for CAD databases.
In *Proceedings of the 17th Design Automation Conference*, pages 50-56. ACM/SIGDA and the IEEE Computer Society DATC, 1980.

[ERCC 78] Edinburgh Regional Computing Centre.
EMAS 2900: Concepts
First edition, 1978.

[ERCC 82] Edinburgh Regional Computing Centre.
EMAS 2900: User's Guide
Third edition, 1982.

[Eswaran 76] Eswaran, K. P., Gray, J. N., Lorie, R. A. and Traiger, I. L.
The notions of consistency and predicate locks in a database system.
Communications of the ACM 19:624-633, 1976.

[Feller 68] Feller, W.
An introduction to probability theory and its applications.
Wiley, New York, London and Sidney, 1968.

[Fernandez 78] Fernandez, E. B., Lang, T. and Wood, C.
Effect of replacement algorithms on a paged buffer database system.
IBM Journal of Research and Development 22:185-196, 1978.

[Fry 76] Fry, J. P. and Sibley, E. H.
Evolution of data-base management systems.
ACM Computing Surveys 8:7-42, March, 1976.

[Goldstein 80a] Goldstein, I. P. and Bobrow, D. G.
Descriptions for a programming environment.
In *Proceedings of the first annual conference of the National Association for Artificial Intelligence*, pages 187-194. August, 1980.

[Goldstein 80b] Goldstein, I. P. and Bobrow, D. G.
Extending object oriented programming in Smalltalk.
In *Proceedings of the 1980 Lisp Conference*, pages 75-81. August, 1980.

[Gordon 79] Gordon, M. J., Milner, A. J. R. G., and Wadsworth, C. P.
Lecture Notes in Computer Science. Volume 78: *Edinburgh LCF*.
Springer-Verlag, Berlin, Heidelberg and New York, 1979.

[Gray 76] Gray, J. N., Lorie, R. A., Putzolu, G. R. and Traiger, I. L.
Granularity of locks and degrees of consistency in a shared data base.
In G. M. Nijssen (editor), *Modelling in data base management systems*, pages 365-394. North-Holland, Amsterdam, New York and Oxford, 1976.

[Gray 79a] Gray, J. N.
Notes on data base operating systems.
In R. Bayer, R. M. Graham, G. Seegmuller (editors), *Operating systems: an advanced course*, pages 393-481. Springer-Verlag, Berlin, Heidelberg, New York, 1979.
Originally published in the series *Lecture Notes in Computer Science*, vol. 60, 1978.

[Gray 79b] Gray, P. M. D. and Bell, R.
Use of simulators to help the inexpert in automatic program generation.
In P. A. Samet (editor), *Euro IFIP 79*, pages 613-620. International Federation for Information Processing, September, 1979.
Participants' edition — proceedings published by North-Holland, Amsterdam.

[Gray 81a] Gray, P. M. D.
Use of automatic programming and simulation to facilitate operations on CODASYL databases.
In M. P. Atkinson (editor), *Database*, pages 345-369. Pergamon Infotech, 1981.

[Gray 81b] Gray, J. *et al.*
The recovery manager of the System R database manager.
ACM Computing Surveys 13:223-242, June, 1981.

[Gremillion 82] Gremillion, L. L.
Designing a Bloom filter for differential file access.
Communications of the ACM 25:600-604, September, 1982.

[Grimson 80] Grimson, J. B.
A flexible database management system for a virtual memory machine.
PhD thesis, University of Edinburgh, 1980.

[Harder 79] Harder, T. and Reuter, A.
Optimisation of logging and recovery in a database system.
In G. Bracchi and G. M. Nijssen (editors), *Data base architecture*, pages 151-168. North-Holland, Amsterdam, New York and Oxford, 1979.

[Hughes 82] Hughes, J. G.
VLSI design aids.
Technical Report, Computer Science Department, University of Edinburgh, 1982.

[Jacobs 82] Jacobs, B. E.
On database logic.
Journal of the ACM 29:310–332, April, 1982.

[Kim 79] Kim, W.
Relational database systems.
ACM Computing Surveys 11:185–211, September, 1979.

[Krasner 81] Krasner, G.
The Smalltalk-80 virtual machine.
Byte 6:300–320, August, 1981.

[Kulkarni 83] Kulkarni, K. G.
Evaluation of functional data models for database design and use.
PhD thesis, University of Edinburgh, 1983.
To be submitted.

[Kung 81] Kung, H. T. and Robinson, J. T.
On optimistic methods for concurrency control.
ACM Transactions on Database Systems 6:213–222, June, 1981.

[Lang 77] Lang, T., Wood, C. and Fernandez, I. B.
Database buffer paging in virtual storage systems.
ACM Transactions on Database Systems 2:339–351, December, 1977.

[Leesley 79] Leesley, M.
Databases for CAD.
Computer Aided Design 11:115–116, May, 1979.

[Lorie 77] Lorie, R. A.
Physical integrity in a large segmented database.
ACM Transactions on Database Systems 2:91–104, March, 1977.

[March 81] March, S. T., Severance, D. G. and Wilens, M.
Frame memory: a storage architecture to support rapid design and implementation of efficient databases.
ACM Transactions on Database Systems 6:441–463, September, 1981.

[Marshall 83] Marshall, R. M.
A Compiler for Large Scale.
CS4 project report, Computer Science Department, University of Edinburgh, June, 1983.

[McGee 77] McGee, W. C.
The information management system IMS/VS.
IBM Systems Journal 16:84–168, 1977.

[Menasce 82] Menasce, D. A. and Nakanishi, T.
Optimistic versus pessimistic concurrency control mechanisms in database management systems.
Information Systems 7:13–27, 1982.

[Michaels 76] Michaels, A. S., Mittman, B. and Carlson, C. R.
A comparison of the relational and CODASYL approaches to data-base management.
ACM Computing Surveys 8:125–151, March, 1976.

[P-E 79] Perkin-Elmer Corporation.
Model 3220 Processor User's Manual
Computer Systems Division, 2 Crescent Place, Oceanport, N. J. 07757, 1979.
Publication number C29-693.

[Rees 80] Rees, D. J.
An associative stack chip.
1980.
Talk presented to the technical discussion group, Computer Science Department, University of Edinburgh.

[Ries 77] Ries, D. R. and Stonebraker, M.
Effects of locking granularity in a database management system.
ACM Transactions on Database Systems 2:233-246, September, 1977.

[Ries 79] Ries, D. R. and Stonebraker, M. R.
Locking granularity revisited.
ACM Transactions on Database Systems 4:210-227, June, 1979.

[Robertson 80] Robertson, P. S.
The IMP-77 Language.
Technical Report CSR-19-77, Computer Science Department, University of Edinburgh, 1980.

[Ross ??] Ross, G. D. M.
An alternative strategy for optimistic concurrency control.
In preparation.

[Senko 77] Senko, M. E.
Data structures and data accessing in data base systems past, present, future.
IBM Systems Journal 16:208-257, 1977.

[Severance 76] Severance, D. G., and Lohman, G. M.
Differential files: their application to the maintenance of large databases.
ACM Transactions on Database Systems 1:256-267, September, 1976.

[Sherman 76] Sherman, S. W. and Brice, R. S.
Performance of a database manager in a virtual memory system.
ACM Transactions on Database Systems 1:317-343, December, 1976.

[Shipman 81] Shipman, D. W.
The functional data model and the data language DAPLEX.
ACM Transactions on Database Systems 6:140-173, March, 1981.

[Sibley 76] Sibley, E. H.
The development of database technology.
ACM Computing Surveys 8:1-5, March, 1976.

[Sidle 80] Sidle, T. W.
Weaknesses of commercial database management systems in engineering applications.
In *Proceedings of the 17th Design Automation Conference*, pages 57-61. ACM/SIGDA and the IEEE Computer Society DATC, 1980.

[Siwiec 77] Siwiec, J. E.
A high-performance DB/DC system.
IBM Systems Journal 16:169-195, 1977.

[Smith 78] Smith, A. J.
Sequentiality and prefetching in database systems.
ACM Transactions on Database Systems 3:223-247, September, 1978.

[Smith 80a] Smith, L. D.
The elementary structural description language.
Technical Report CSR-53-80, Computer Science Department, University of Edinburgh, 1980.

[Smith 80b] Smith, L. D.
SIM - a gate level simulator.
Technical Report CSR-60-80, Computer Science Department, University of Edinburgh, 1980.

[Stonebraker 76]
Stonebraker, M., Wong, E., Kreps, P. and Held, G.
The design and implementation of INGRES.
ACM Transactions on Database Systems 1:189-222, September, 1976.

[Stonebraker 80]
Stonebraker, M.
Retrospective on a database system.
ACM Transactions on Database Systems 5:225-240, June, 1980.

[Stonebraker 81]
Stonebraker, M.
Operating system support for database management.
Communications of the ACM 24:412-418, July, 1981.

[Stonebraker 83]
Stonebraker, M. *et al.*
Performance enhancements to a relational database system.
ACM Transactions on Database Systems 8:167-185, June, 1983.

[Strickland 82] Strickland, J. P., Uhrowczik, P. P. and Watts, V. L.
IMS/VS: an evolving system.
IBM Systems Journal 21:490-510, 1982.

[Sussman 72] Sussman, G. J. and McDermott, D. V.
From PLANNER to CONNIVER - a genetic approach.
In *AFIPS Conference Proceedings*, volume 41 part II, pages 1171-1179. American Federation of Information Processing Societies, December, 1972.

[Tate 81] Tate, B. A.
RBASE: a relational database management system on EMAS.
Technical monograph no. 1, Edinburgh Regional Computing Centre, June, 1981.

[Taylor 76] Taylor, R. W. and Frank, R. L.
CODASYL data-base management systems.
ACM Computing Surveys 8:67-103, March, 1976.

[Traiger 82] Traiger, I. L.
Virtual memory management for database systems.
ACM Operating Systems Review 16:26-48, October, 1982.

[Tsichritzis 76] Tsichritzis, D. C. and Lochovsky, F. H.
Hierarchical data-base management: a survey.
ACM Computing Surveys 8:105-123, March, 1976.

[Tukey 77] Tukey, J. W.
Exploratory data analysis.
Addison-Wesley, 1977.

[Verhofstad 78] Verhofstad, J. S. M.
Recovery techniques for database systems.
ACM Computing Surveys 10:167-195, June, 1978.

[von Mises 64] von Mises, R.
Mathematical theory of probability and statistics.
Academic Press, New York and London, 1964.

[Warman 79] Warman, E. A.
Computer aided design problems for the 80's.
In P. A. Samet (editor), *Euro IFIP 79*, pages 499-502. International Federation for Information Processing, September, 1979.
Participants' edition — proceedings published by North-Holland, Amsterdam.

[Wiederhold 77] Wiederhold, G.
Database design.
McGraw-Hill Book Co., New York, 1977.

[Xerox 81] The Xerox Learning Research Group.
The Smalltalk-80 system.
Byte 6:36-48, August, 1981.

[Zhintelis 79] Zhintelis, G. B. and Karchyauskas, E. K.
Software design for CAD with repeated editing.
Programming and Computer Software 5:285-289, July-August, 1979.
Translated from *Programmirovanie* 4, 1979.

Appendix A
The Rbase User Interface

This appendix is intended as an introduction to the basic facilities provided by the *Rbase* interactive query facility in so far as this is necessary to understand the tests of chapter 4. It is not intended as a complete user's guide to the *Rbase* system, for which see [Tate 81].

A. 1 Items

The database consists of a collection of *items*, each of which has an *identifier* and a *value*. An *identifier* (ID) may be *simple* or *compound*. A simple ID is any string, excluding simple and compound ID delimiters. A compound ID consists of a left compound ID delimiter followed by from one to five IDs and terminated by a right compound ID delimiter. By default the left and right compound ID delimiters are "[" and "]" respectively, and the simple ID delimiters are space or newline.

Some examples of simple IDs are:

```
VAX
CS4
#7/$@243c$%dw
```

Some examples of compound IDs are:

```
[name Caesar]
[toy [ball red]]
[CSDEPT GDMR [ROOM 2417] [PHONE 2788]]
```

A *value* is any string. Some examples are:

```
Computer Science
480 Kg per square metre
```

Note that certain combinations of characters may be given specific interpretations by *Rbase*: "?" may be interpreted as the start of an actor (see below), and the value *UNDEF* will be interpreted to mean that the value is undefined.

A. 2 Actors

It is possible to specify a subset of the items in the database by giving a specification of the IDs and values of items required. This is done using *ACTORS*. Specification of items for possibility lists or individual items can be done using explicit item IDs, actors or a combination of both to specify matching item IDs, and using explicit strings or actors to specify matching values. The actors currently provided are as follows:-

??	matches anything.
?A .. ?Z	allows matches to be performed to check for common sub-items. An actor variable does not keep its value between matches.
?[NOT id]	matches anything that does not match ID.
?[ANY id1 id2 ...]	matches anything that matches any one of ID1, ID2, ... Up to four IDs are allowed.
?[ALL id1 id2 ...]	matches anything that matches all of ID1, ID2, ... Up to four ids are allowed.
?[LENGTH *n*]	matches items with length *n*, which must be an explicit integer. *n* = 0 for simple IDs, *n* = 1 .. 5 for compound ids.
?[> string]	string relations on simple ids. Also allowed are ?[< string] and ?[# string].
?[SUBSTRING xxxxx]	matches simple ids with xxxxx anywhere.

Although values of items are not part of the *Rbase* storage structure, all the above actors except LENGTH may be used to specify matches for the values of items.

A. 3 The interactive query facility

The interactive query system will respond with the prompt "Query:" when it is ready to accept commands. The following are available:

STORE(s, v)	give item with ID s a value v.
GET ONE(s, v)	get an item matching s and v and print its identifier and value.
GET ALL(s, v)	as above for all items matching s and v.

VALUE(s)	gets the values of one item matching s.
MODIFY ALL(s, v, newv)	modify all items matching s and v to have the new value newv.
ERASE ALL(s, v)	erase all items matching s and v.
CREATE(name)	creates and initialises a new database called "name".

OPENDB(name, password, mode)

opens the database of the given name. The password parameter must match the database password (if it has been set). Mode should be one of

- *READ*, when the database will be opened in read-only mode
- *WRITE*, when the database will be opened in read/write mode
- *TEMP*, when the database will be opened so as to permit writing, though any updates will be thrown away when the database is closed

CLOSE DB	closes the data base and returns to *VAX/VMS*.
BRACKETS	sets the parameter delimiters to be "(" and ")"
NOBRACKETS	resets the parameter delimiters to be space and newline

String quotes are not needed around identifiers or values. Parameters should not contain commas or right brackets as these are used as delimiters (optionally a space character may be used in place of '(' and ')' as delimiters, in which case the commands must be given as a single word — see *brackets* and *no brackets* above).

Appendix B
The Interfaces to the Block-based Implementation

B.1 Overview

The system is constructed as a number of modules, used as *Imp* <u>include</u> files,[1] *viz* a system configuration file, a file of system utilities, the file system interface and the differential file manager proper.

Mounting the system under a new operating system requires that the file system interface module be written, and possibly that modifications be made to the system utilities file and the system configuration file. An experienced systems programmer should be able to complete these tasks in a day, probably in an afternoon. The only assumption made is that the operating-system and machine configuration supports 32-bit integers and direct (random) access files (*DA files*).

B.2 Differential file manager

The user perceives the differential files in use as *virtual DA* files; these are accessed by name only when they are created or opened, at which time a correspondence is set up between the virtual file and a "slot number" by which the file can be referenced in subsequent operations. The interface is given in figure B-1. In all cases write mode is indicated by setting the sign bit of the slot number.

Open DB Open a virtual file and associate the given slot number with it. If the file is already open then it is automatically shared among all the slots which refer to it; otherwise if the file is a differential file then the differential part is opened and a recursive call is made to open the static part, the name of which is stored in the header block of the differential part. Thus opening

[1]Six segments would be required on the *3220* under *Mouses* if the modules were to be compiled separately and linked (dynamically) at run-time. As a further seven segments are required by the subsystem, the diagnostics program and the user stack and heap, this would mean that only three segments would be available for the user program, an unacceptably low number.

Figure B-1: User interface procedures

```
routine OPEN DB   (integer slot, string(63) file)

routine CREATE DB (integer slot, with,
                   string(127) file)

routine CLOSE DB  (integer slot)

routine MERGE DB  (integer slot)

record(*)namefn GET BLOCK (integer slot and block)

record(*)namefn NEW BLOCK (integer slot and block)

routine BLOCK RESIDENCY (integer slot and block, mode)
```

the virtual file on the leaf of any tree will cause the leaf itself and all intermediate files up to and including the root to be opened (or shared if opening another virtual file has already caused them to be opened).

Create DB Given the slot number of an already-open virtual file (*with*), create a new differential file layer with the given name and associate the given slot number with it.

Close DB Close the virtual file associated with the slot number. Decrement the reference counts of all the physical files on the path back to the root file, and if any becomes zero close the physical file.

Merge DB Merge the (topmost) differential file of the virtual file associated with the given slot into its static part, alter the slot correspondence to refer to the merged static part and then delete the differential file (N.B. the reference count of the static part must be one for this to be allowed, i.e. no other virtual file can refer to the static part).

Get block Bring the requested block into the disc cache and return its address. The block number is coded up as

```
slot << 24 ! block.
```

Write mode is indicated by setting the sign bit of the parameter.

New block Allocate a new block in the virtual file, extending it if necessary, and return the address of a block in the disc cache which has been mapped onto it. The block and slot numbers are coded up as for *Get block*.

Block residency The default behaviour of the disc cache is to reuse the least recently used block, writing out or forgetting the previous contents as appropriate. As described in section 6.7, however, it is sometimes useful to be able to vary this behaviour; this procedure provides such a facility. The block and slot numbers are coded up as for *Get block*. The mode parameter is used as follows: *negative* indicates "toss immediately"; *zero* indicates default (least recently used) behaviour; *positive* indicates lock in cache (indefinitely, until explicitly released by another call on this procedure.

B.3 File system interface

This module provides a standard interface to the host filing system, attempting to hide any peculiarities from higher levels. The interface spec is given in figure B-2.

Figure B-2: File system interface procedures

```
routine OPEN FILE   (string(*)name file,
                     integer slot, mode,
                     integername file size)

routine CREATE FILE (string(*)name file,
                     integer slot, blocks)

routine CLOSE FILE  (integer slot)

routine DELETE FILE (integer slot)

routine EXTEND FILE (integer slot, blocks,
                     integername new size)

routine READ BLOCK  (integer slot, block,
                     record(*)name buffer)

routine WRITE BLOCK (integer slot, block,
                     record(*)name buffer)

string(63)fn EXPAND FILE (string(63) file)

string(31)fn UNIQUE FILE
```

Operations are again performed in terms of a logical slot which is used by higher levels to refer to files; the file system interface is required to translate these to the naming convention in force in any particular configuration.

The interface procedures perform the following functions:

Open file	Given a filename, a slot number and an access mode open the file. Return the file size (in blocks) and the filename fully expanded according to the conventions in force. Mode is zero for read-only, non-zero for read/write.
Create file	Given a slot number and a size (in blocks) create a file of (at least) the required size. Return the fully expanded filename.
Close file	Given a slot number close the corresponding file.
Delete file	Given a slot number delete the corresponding file.
Extend file	Given a slot number and an extend amount (in blocks) extend the file. Return the new size for future reference (some operating systems are generous in extending files!).
Read block	Given a slot number, a block number and the address of a buffer read in the requested block into the buffer.
Write block	Given a slot number, a block number and the address of a buffer write the buffer out to the requested block.
Expand file	Given a filename expand it according to the conventions in force and return the fully expanded name.
Unique file	Return the name of a unique (non-existent) file.

B.4 System configuration

The system is configured by means of an *Imp77* include file. The configuration file used for the performance tests is shown in figure B-3.

The significance of these constants is as follows:

max file slot	This is the maximum number of physical files which the system is allowed to have open at any time. This figure should not be greater than the operating-system permitted maximum, as otherwise the system may fail with unhelpful error messages, but need not be equal to it (under *Mouses* a total of about 80 file slots are shared among all the users of the system). In the current implementation this figure should not be greater than 32.
block size	This is the physical size of disc blocks (in bytes). This figure **must** be equal to the hardware-defined figure for the operating system in question.
max logical slot	This is the maximum number of virtual files which the user may have open. This figure may be greater than the corresponding physical open-file limit as the automatic file-sharing mechanism built into the system will mean that

Figure B-3: System configuration file used for test runs

```
! system configuring constants (Mouses test runs)

constinteger max file slot      = 10;   ! slots
constinteger block size         = 512;  ! bytes
constinteger max logical slot   = 12;   ! slots
constinteger cache size         = 24;   ! blocks
constinteger cache allocated    = 48;   ! blocks
constinteger initial file size  = 64;   ! blocks
constinteger extend size        = 32;   ! blocks
constinteger default t blocks   = 4;    ! blocks

! other constants

constrecord(*)name nil == 0
constinteger infinity = x'7FFFFFFF'

endoffile
```

several logical files may map onto one physical file. There is no real reason for being unnecessarily restrictive with this figure (except that the limit of 32 applies, as it does for the corresponding physical-file limit), as only 4 bytes are allocated statically for each permitted file.

cache size and *cache allocated*
These two define the size of the disc block cache, the former being the default number of blocks which the system is permitted to use, the latter being the number of blocks allocated at system initialisation. These two are distinct to permit the effective cache size to be varied as a test parameter.

initial file size When a new file is created this figure gives the amount of disc space to allocate to it (in blocks).

extend size When an existing file is extended this figure gives the amount by which it should be extended (in blocks).

default t blocks This is the default number of translation blocks to allocate in a differential file (the actual number allocated may be varied by setting the value of the external integer *translation blocks* before creating the differential file).

The remaining two constants are not really configuration constants; they require to be defined for all the system modules, however, so it seems reasonable to include them here.

B. 5 System utilities

The system utilities file is operating-system independent for the most part, although it may be possible to optimise the system on some machines by making use of specific hardware- or language-support facilities. The module provides a bulk-move procedure, a case-flipping procedure, bitmap manipulation procedures, the error-reporting procedure and access to a heap.

Appendix C
The User Interface to the Shrines Utility

This appendix describes the user interface to the *Shrines* utility, originally conceived and part-implemented by Paul MacLellan. The user is provided with a set of procedures with which to open, close, create and modify a shrine. Any number of shrines may be open at any time. While a shrine is open it may be accessed as though it were a "normal" area of virtual memory. The explicit interface procedures are listed in figure C-1, with a further implicit interface being by way of the secondary exception handler used by the shrine manager to trap access violations to as-yet unmodified pages.

Figure C-1: Shrines user interface procedures

```
integerfn OPEN SHRINE (string(127) filename,
                       integername lowest, highest)

integerfn CLOSE SHRINE

integerfn CHECKPOINT SHRINE

integerfn CHECKPOINT AND CLOSE SHRINE

integerfn CREATE SHRINE (string(127) filename,
                         integer size, extra)

integerfn EXTEND SHRINE (string(127) filename,
                         integer size, extra)

integerfn EXTEND OPEN SHRINE (integer anyaddr,
                              integer size, extra,
                              integername lowest,
                                          highest)

integerfn SHRINE FREE SPACE  (integer anyaddr,
                              integername free)
```

In all cases where a shrine is accessed explicitly via one of these procedures, a completion status is returned as the function result, while if an error occurs while the shrine is being accessed implicitly as part of a process's virtual memory the *VMS* signalling mechanism [DEC 82c] is used. In both cases standard *VMS* system error codes are returned.

The interface procedures are used as follows:-

OPEN SHRINE The requested shrine is opened and mapped into virtual memory, the start and finish addresses being returned to the user. The filename parameter may optionally be specified as a name variable, when the fully expanded shrine name will be returned in it.

CLOSE SHRINE The shrine is closed (without checkpointing). All modifications made since it was either opened or last checkpointed are lost.

CHECKPOINT SHRINE
The shrine file on disc is made consistent and the root blocks are exchanged. All updates previously made will be preserved on disc. Any updates subsequently made must be preserved by a further checkpoint operation.

CHECKPOINT AND CLOSE SHRINE
A combination of the above two operations. It is, however, more efficient than calling them separately since there is no need to reinitialise the shrine after checkpointing.

CREATE SHRINE A new shrine is created of size <size> with <extra> additional space for pending blocks. As for *open shrine*, the filename parameter may be specified as a name.

EXTEND SHRINE An currently-open shrine, which must not be open, is extended so as to be of size <size> and have <extra> additional space for pending blocks.

EXTEND OPEN SHRINE
An already existing shrine is extended so as to be of size <size>, with <extra> additional space for pending blocks. The shrine will be remapped, the new bounds being returned in <lowest> and <highest>. Since more than one shrine may be open it is necessary to indicate which is to be extended by specifying the address of any byte in the shrine.

SHRINE FREE SPACE
This is provided to aid the user program in deciding when to extend a shrine. The shrine is selected by providing the address of any of its bytes in <anyaddr>. The number of blocks currently unused is returned in <free>.

If more than one shrine is open it is necessary to indicate which is to be checkpointed and/or closed. This is done by specifying an optional parameter to the three checkpointing and closing procedures, *viz* the address of any byte in the shrine. This parameter is obligatory for *Extend Open Shrine* and *Shrine Free Space*.

In addition to the basic package there is an analysis program which may be used to obtain (static) space utilisation statistics. This program is useful in tuning the size of a shrine and the amount of extra space for a particular application. To use it, it is first necessary to define a *DCL* symbol

```
    $ ShrineAnal :== $U0:[GDMR.Shrines]Analyse
```

It may then be invoked by a command of the form

```
    $ ShrineAnal <filename>
```

where <filename> is the name of the shrine to be analysed. For example

```
    $ ShrineAnal TEST
```

might produce the following output:

```
Analysis of shrine U0:[GDMR.SHRINES]TEST.SHR;1
Total file size 244 blocks, mapped at 0003A600..00058DFF
Created on Thursday 2nd at 6.56pm
Last accessed on Thursday 2nd at 6.59pm

Current root at offset 1, epoch 4
Shrine size 192 blocks (78.7% of file)
1 map page (0.4% of file)
192 data pages (100.0% of shrine, 78.7% of file)

Previous root at offset 0, epoch 3
Shrine size 192 blocks (78.7% of file)
1 map page (0.4% of file)
192 data pages (100.0% of shrine, 78.7% of file)

0 shared map pages (0.0% of map)
191 shared data pages (99.5% of shrine, 99.5% of data)
48 blocks unused by either root (19.7% of file)
```

Appendix D
VAX/VMS System Parameters

This appendix gives a brief description of the VAX/VMS system generation parameters which affect process paging. For a fuller discussion see [DEC 81, DEC 82e].

Both paging and swapping are used in the memory management system. Swapping is done on a system-wide basis, while paging takes place on a per-process basis. While it is resident, each process is guaranteed that it may extend its working set up to its authorised limit; in addition it may extend its working set beyond this guaranteed quota by "borrowing" pages from the system, subject to there being adequate slack available. Should there be insufficient space for a process to expand to its guaranteed limit, the swapper will swap out one or more processes from the balance set in order to make room.

As a process executes, pages migrate between the process's working set and the free page and modified page lists. Page replacement from the working set is organised on a first-in-first-out basis; when a page is removed it is placed on the end of the free page list or the modified page list, as appropriate. If there is a subsequent fault for that page it is recaptured from the appropriate list and returned to the user's working set. When the size of the free list falls below a pre-defined threshold, or when the size of the modified list grows beyond a pre-defined threshold, the modified page writer is awakened to write part of the modified list out to the page-file.

The size of a process's working set is determined by three parameters, defined for that process either in the system *User Authorisation File* if the process is an interactive user or a batch job (further modified by the characteristics of the batch queue in the latter case), or by the arguments to the $CREPRC system service in the case of a detached process. They are:

WSDefault	During image winddown a process's working set will be reduced in size to that specified by this parameter.
WSQuota	This defines the working set size which a process is guaranteed while it is executing.
WSExtent	Subject to there being sufficient slack in the system (see below) a process may increase its working set up to this value. It is not guaranteed to be able to do this, however.

The system-wide parameters, defined during system initialisation, control the sizes of the free and modified lists, determine when a process may be allowed to borrow beyond its WSQuota, and define paging cluster factors. They are:

BorrowLim	Defines the minimum number of pages which must be on the free list before a process is allowed to grow beyond its WSQuota.
FreeGoal	Defines the number of pages desired on the free list when action is taken as a response to the size of the free list falling below FreeLim.
FreeLim	Defines the minimum number of pages which must be on the free list. The system will write pages from the modified page list or swap out or reduce process working sets in order to maintain this figure.
GrowLim	Sets the number of pages which must be on the free list before a process which is already over its WSQuota is allowed to extend its working set even more.
MPW_HiLimit	Sets the upper limit for the size of the modified page list. If this is exceeded then the system will begin to write out pages, transferring them to the free list when they have been written.
MPW_LoLimit	Defines the lower limit for the modified page list size. Writing out of the list will stop when the number of pages falls below this figure.
MPW_Thresh	Sets a lower bound on the number of pages which must be on the modified list before the swapper will write modified pages in order to acquire pages for the free list.
MPW_WaitLimit	Defines the number of pages on the modified page list which will cause a process to wait until such time as the modified page list is written out. (This acts as a brake on processes producing large numbers of modified pages.)
MPW_WrtCluster	Defines the number of pages which will be written out to the page file from the modified page list in a single I/O operation.
PFCDefault	Sets the default number of pages read from disc when a page fault occurs.

Appendix E Microcode Assistance

Although the algorithms used in a system which implements experiments are sufficiently complex that coding in a high-level language is desirable to ensure comprehensibility and maintainability, nevertheless there are a number of areas where microcode assistance might be beneficial to performance. Some more modern machine architectures may already incorporate functions which would otherwise be candidates for microcoding, such as the bulk-move MOVC3 and MOVC5 *VAX* instructions [DEC 77b]. In such cases the gain in performance from microcode assistance may be marginal [Stonebraker 83]. The following examples serve to illustrate the sort of sub-problem which would be amenable to a microcode approach:

- There are two critical areas in the implementation described in chapter 5 where the increase in speed available from microcode assistance would be of most benefit. These are:

  1. The filter. This section is executed every time a translation is required in order to determine which of the parts of the virtual file contains the required block. Clearly, any increase in the efficiency with which this is carried out could not fail to result in an improvement of system performance.

  2. The cache. As table 6-9 showed, increasing the cache size did not automatically result in an increase in performance, as the increasing CPU costs tended to counter the reduction in I/O costs. A microcode cache algorithm would reduce the overheads incurred in searching the cache for blocks.

- If experiments were to be introduced to the *PS-algol* system then the pointer translation algorithm described in section 2.4.5 would be an obvious candidate for microcoded assistance. The algorithms involved here could also be used for cache maintenance.

- A microcoded translation scheme, which would, at first sight, bring great benefits would be less easy to implement, as the data structures involved will not, in general, be entirely resident. If the micro-architecture is such that a macroinstruction may be suspended while a trap is taken to a user-supplied macro-routine and then resumed, however, then such a scheme would be possible.

The filter algorithms are, in many ways, very similar to those used to manipulate "normal" bitmaps, and hence any instructions which are provided for use in the latter case would also be beneficial in the former. The relevant instructions on the *3220* are TBT (test bit) and SBT (set bit),[1] while the relevant *VAX* instructions are BBS (branch on bit set), BBC (branch on bit clear) and BBSS (branch on bit set and set). In the case of the last of these, by making the branch destination the next instruction in sequence we obtain the effect of a bit-set instruction.

The cache algorithm used in the implementation described in chapter 5 was based on the "least recently used" principle, using a global sequence number as a timestamp. The requirement here is that the microcoded instruction take as its input the address being searched for and the size of the cache, and provides as output the location in the cache of either the address (if found) or the oldest slot. In the former case, the sequence number of the cache slot would be set to the current sequence number, which would then be incremented, while in the latter case the cache manager would require to write out the old data (if it had been modified) and read in the new data.

Special instructions for maintaining an associative stack, such as would be required for the tagged implementation of experiments with the persistent programming model (see section 2.4.5 on page 21) or the context mechanism of *PIE* [Goldstein 80b], could also be used as an alternative cache strategy. Here, an tag is presented, together with a list of possible values: the microcode for the instruction searches the list for the required tag, moving the entry to the head of the list and returning the corresponding value if the tag is found. If the tag is not found, another instruction would push a tag/value pair onto the stack, returning as a result the one (if any) which was pushed off the bottom. As an alternative to microcode assistance, special purpose hardware (such as a VLSI associative stack chip [Rees 80]) could be used. This latter possibility has particular relevance to "meccano-kit" machines based on microprocessor technology, such as the Computer Science Department's *advanced personal machine* [Dewar 83].

Facilities were not available to experiment with microcode assistance, as the Computer Science Department's *VAX* is required to provide a continuous service and so could not be used to debug experimental microcode, while the Data Curator Project's *3220* was not fitted with the writeable control store option. The arrival of a number of ICL *PERQ*s with writeable control store will, however, mean that this area becomes amenable to investigation.

[1]As the bits in the filter bit-table will never be cleared, the RBT (reset bit) and CBT (complement bit) instructions are not relevant.