

PERSISTENT MEMORY:

A Storage Architecture for Object-Oriented Database Systems

Satish M. Thatte

Artificial Intelligence Laboratory, Computer Science Center
Texas Instruments Incorporated

P.O. Box 226015, M/S 238, Dallas, TX 75266

Abstract

Object-oriented databases are needed to support database objects with a wide variety of types and structures. A persistent memory system provides a storage architecture for long-term, reliable retention of objects with rich types and structures in the virtual memory itself. It is based on a uniform memory abstraction, which eliminates the distinction between transient objects (data structures) and persistent objects (files and databases), and therefore, allows the same set of powerful and flexible operations to be applied with equal efficiency on both transient and persistent objects from a programming language such as Lisp or Prolog. Because no separate file system is assumed for long-term, reliable storage of objects, the system requires a crash recovery scheme at the level of the virtual memory, which is a major contribution of the paper. It is expected that the persistent memory system will lead to significant simplifications in implementing applications such as object-oriented databases.

1 Introduction

Many new applications, such as computer-aided design (CAD) in engineering disciplines, multi-media information systems, and expert database systems require databases that can support objects of a wide variety of *types* with the ability to express complex *relationships* among objects [Hartzband85]. The type of an object refers to a template used for creating instances of that type. The structures of objects reflect the relationships among objects. Object-oriented programming and object-oriented systems have much to offer to develop databases with an "object-oriented" view [OOS85]. Object-oriented databases can greatly benefit from object-oriented programming systems, such as the Flavors system on Lisp machines and the Smalltalk system, widely used in the symbolic computing and artificial intelligence community. These systems provide sophisticated and rich abstraction mechanisms to define abstract types and to hide the implementation details of objects. Objects have a well-defined operational interface to its users. Powerful inheritance mechanisms are available to build higher levels of abstractions, i.e., higher-level objects can inherit the properties of other objects. Large systems

can be divided naturally into coherent parts, which can be developed and maintained separately. Also, specifying redundant information can be eliminated.

The advent of automatically managed, garbage-collected virtual memory was crucial to the development of today's object-oriented systems based on object-oriented programming languages. Such a storage architecture removes the burden of storage management (storage allocation and deallocation, garbage collection, compaction, paging) from the programmer. In addition the storage systems allow efficient representation and sharing of objects with a wide variety of types and structures via sophisticated pointer structures. No analogous storage organization has yet been developed in the domain of *persistent* objects managed by today's file or database systems. As a consequence, the programmer is forced to flatten rich structures of objects resident in virtual memory before the objects can be stored in a file system or conventional database. This task puts a great burden on the programmer and adversely affects system performance.

We believe that the architecture of a storage system will play a crucial role in the ease and efficiency with which object-oriented database systems can be developed and used. The storage architecture required by persistent memory ties the notion of object *persistence* to the garbage collection process; an object persists independent of its type or the storage medium on which it resides, so long as it cannot be garbage collected. Thus, persistent memory extends the concept of automatically managed, garbage collected virtual memory to a storage architecture that also includes database objects. In a persistent memory, both transient and persistent objects of a given type have identical representations.

Outline of the paper: Section 2 presents a critique of existing approaches to implementing object-oriented databases. Section 3 briefly describes our approach based on persistent memory. The persistent memory system is based on a *uniform memory abstraction*, which eliminates the distinction between transient and persistent objects, and therefore allows the same set of powerful and flexible operations to be applied with equal efficiency on both transient and persistent objects from a programming language such as Lisp or Prolog, without requiring a special-purpose database language. The uniform memory abstraction is presented in Section 4. The storage architecture, as presented by persistent memory, supports long-term, reliable retention of richly structured objects in the virtual memory itself.

without resorting to a file system. Therefore, implementation of the architecture requires a crash recovery scheme at the level of virtual memory. A crash recovery scheme for the persistent memory is presented in Section 5. It is based on an efficient checkpointing and roll back scheme. Resilient objects are persistent objects that survive system crashes and shutdowns, even if they are created or updated after the last checkpoint operation. Section 6 describes the management of resilient objects. A persistent object manager is implemented on top of the persistent memory to name and manage persistent objects. Section 7 briefly describes the functions of such a manager.

As the title of this paper reflects, the paper concentrates on the storage architecture for object-oriented databases, not on the database manager itself. It provides only a brief description of a database manager that can be implemented on top of the persistent memory. We are in the process of implementing a prototype of the persistent memory on a TI Explorer Lisp machine. Section 8 briefly describes the prototype. Our current work is focused on implementing the persistent memory on single workstations. We have initiated a research effort to extend the concept of persistent memory in a network-based distributed computing environment so that multiple users can share persistent objects on multiple machines. Section 9 presents our research direction on this challenging research problem. Section 10 concludes the paper.

2 Critique of Existing Approaches to Object-Oriented Databases

Conventional database systems (network, hierarchical, or relational models) are not adequate to support the needs of object-oriented databases because they do not have the necessary diversity and richness of types and structures of objects. Realizing these shortcomings of conventional databases, the research community has followed a number of different approaches to develop databases to support a richer variety of object types and structures. These efforts can be classified into two broad classes and are briefly reviewed below.

1. Extension of conventional database models: The database community has extended conventional database models with richer data types. The INGRES relational database has been extended to enhance its applicability to engineering problems [Stonebraker84]. Support for pictorial databases has been implemented on top of and by extending relational and hierarchical databases in the SDMS [Herot80] and VGQF [McDonald81] systems. The MAPS system is a cartographic data manager with explicit support for the display of spatial knowledge [McKewon84].

Although these systems have contributed to the understanding of the problems associated with supporting a diversity of data types, they all have involved grafting additional functionality on top of an existing model. Therefore, many capabilities, such as recursive procedures, or functions as true first-class objects [Abelson85], are awkward,

difficult, or inefficient (due to interpretive overhead) to implement in these extended databases. As a consequence, the applicability of the resultant systems has been limited [Hartzband85].

2. Persistent object systems: Recently many researchers [Atkinson83], [Atkinson84], [Cockshott84], [Mishkin84], [Butler86] have followed the approach of making objects created and manipulated by programs *persistent*. Butler has developed a scheme to make Lisp objects persistent on top of a conventional database (INGRES) [Butler86]. The programmer must declare that the value of a variable is persistent and all the fetching and updating is handled automatically. The low-level Lisp functions that manipulate lists (e.g. CAR, CDR, ...) have been altered to examine their arguments to check for persistence. To partially overcome the retrieval costs of fetching persistent objects from the INGRES database, objects are buffered in virtual memory. Persistent Object Management system [Cockshott84] is an ongoing research effort to produce an extension of Algol that provides automatic access to a database system. Mishkin's OM system extends a dialect of Lisp to encompass persistent Lisp objects [Mishkin84].

The designers of these systems rejected the use of conventional databases for their limited repertoire of data types and structures, and built new database systems to handle data types needed by their programming languages and extended the heap facilities of their respective programming languages to provide access to heaps that reside in their databases. They also built new database primitives that are appropriate to operate on the data structures in persistent heaps. The advantage of this approach is that the system can provide only the facilities needed by the particular language. No overhead is paid for features not useful to the language.

In all these persistent object schemes, persistent and transient objects reside in two different storage organizations; transient objects reside in the virtual memory, and persistent objects in a general-purpose database [Butler86], or in a special-purpose database [Mishkin84], [Atkinson84], [Cockshott84]. Therefore, operations on persistent objects are invariably slower than on transient objects. There is a large overhead to access persistent objects because their pointers must be dereferenced by software, taking several machine cycles. In addition, buffer management is necessary to reduce the cost of fetching persistent objects.

3 Our Approach to Persistent Objects

Symbolic computers, such as the TI Explorer¹, Symbolics 3670, and Xerox 1108, are perhaps the best tools available today to implement applications that need a rich variety of object types and structures because of their support for rich knowledge representation and inference techniques, object-oriented programming languages, and integrated program development environments. In these computers, objects representing knowledge exhibit a *structure*

defined by pointers connecting objects. This structure is usually complex and dynamic, i.e., it changes rapidly. All processes and objects share a single virtual address space [TI85a]. Sharing of objects via pointer structures allows efficient and flexible representation of knowledge. The representation is also processed most efficiently by the machine because it is defined in the machine architecture, and hence is directly interpreted by hardware or microcode. The pointers to objects serve as *names* that can be passed as procedure parameters, returned as procedure results, and stored in other objects as components. A high proportion of data is pointers to other data and structures. This storage model requires automatically garbage collected memory – a feature supported by Lisp machines.

Our approach to object persistence in a symbolic computing environment is based on a fundamentally different paradigm [Thatte85], [Thatte86]. We would like to make objects persistent in the virtual memory itself. The literature on persistent² memory dates back to 1962, when Kilburn [Kilburn62] proposed *single-level* storage, in which all programs and data are named in a single context. Saltzer [Saltzer78] proposed a *direct-access* storage architecture, where there is only a *single* context to bind and interpret all objects. Traiger [Traiger82] proposed mapping databases into virtual address space. It seems that simple data modeling requirements of computer applications of that time discouraged productization of these proposals partly because they are much more difficult to implement than the conventional virtual memory and database systems. We strongly believe that these proposals must be revived and adapted to the needs of object-oriented databases if we are to support their demanding requirements of data modeling and long-term storage of objects with rich types and structures.

The MIT MULTICS system [Bensoussan69] and the IBM System/38 [IBM38] have attempted to reduce the storage dichotomy. However, both have major shortcomings for symbolic computing. All persistent information is in files. A file mapped into the address space of a process cannot hold a machine pointer to a file mapped in the address space of a different process. Thus, sharing of information among different processes is more difficult than with Lisp machines. Furthermore, there is no automatic garbage collection, which is essential for supporting symbolic languages.

The persistent memory is based on a *uniform memory abstraction*. In this abstraction, an object persists as long as it can be prevented from being garbage collected. The abstraction is implemented on a single, large virtual address space to support large knowledge-based applications. The concept of persistent memory, however, does not depend on the address space size. A major contribution of this paper is a recovery scheme at the level of the virtual memory itself, i.e., a *recoverable* virtual memory. A recov-

erable virtual memory is essential because no separate file system is assumed for the purpose of recovering permanent data. The implementation of recoverable virtual memory is based on an efficient checkpointing scheme that incrementally captures the entire state of the machine, and a roll back scheme that rolls back the machine state to the last checkpoint following a system crash.

Our approach is illustrated in Figure 1. Successively more powerful abstractions are created on top of the physical memory resources. The first layer of abstraction is a recoverable virtual memory. A garbage collector runs on top of it to reclaim space occupied by inaccessible or garbage objects. Discussion on garbage collection is outside the scope of this paper. A good treatment can be found in [Moon84], [McEntee86].

Persistent objects created after the last checkpoint will not survive a system crash. Similarly, the checkpointed state will not reflect changes made to persistent objects after the checkpoint but before the crash. We define a resilient object as one which can not only survive beyond the lifetime of a program that created it, but can also survive system crashes. Resilient objects require the notion of atomic actions or transactions for their implementation. Thus, resilience is a stronger property than persistence. Not all applications require resilient objects.

Thus, as shown in Figure 1, the persistent memory defines a storage architecture to support applications that combine the strengths of AI and database technologies. It should be mentioned that the proposed system is *not* intended to completely eliminate the need for a file system. A file system will be needed as an archival medium to store objects for which there is no space in persistent memory. A file system will also be needed to support dismountable or removable storage media.

4 Uniform Memory Abstraction

The uniform memory abstraction is an *abstraction* of a persistent memory system, i.e., it defines the *architecture* of a storage system that manages both transient and persistent memory objects *uniformly*. As an abstraction it defines only *what* the external characteristics of the persistent memory should be, and *not how* to implement a system with these characteristics. As shown in Figure 2, in the uniform memory abstraction a processor views memory as a set of variable-sized blocks or objects interconnected by pointers. Each memory object consists of one or more memory words, which are stored in *consecutive* virtual addresses. Pointers are typically implemented as virtual addresses. The notion of memory objects in the uniform memory abstraction corresponds to objects used in high-level programming languages, such as numbers, booleans, characters, strings, Lisp CONS cells, arrays, vectors, records, procedures, or environments. These language-level objects can be implemented using one or more memory objects interconnected

¹Explorer is a trademark of Texas Instruments Incorporated.

²In the literature, terms such as *permanent*, *stable*, *direct-access*, or *single-level* storage are also used.

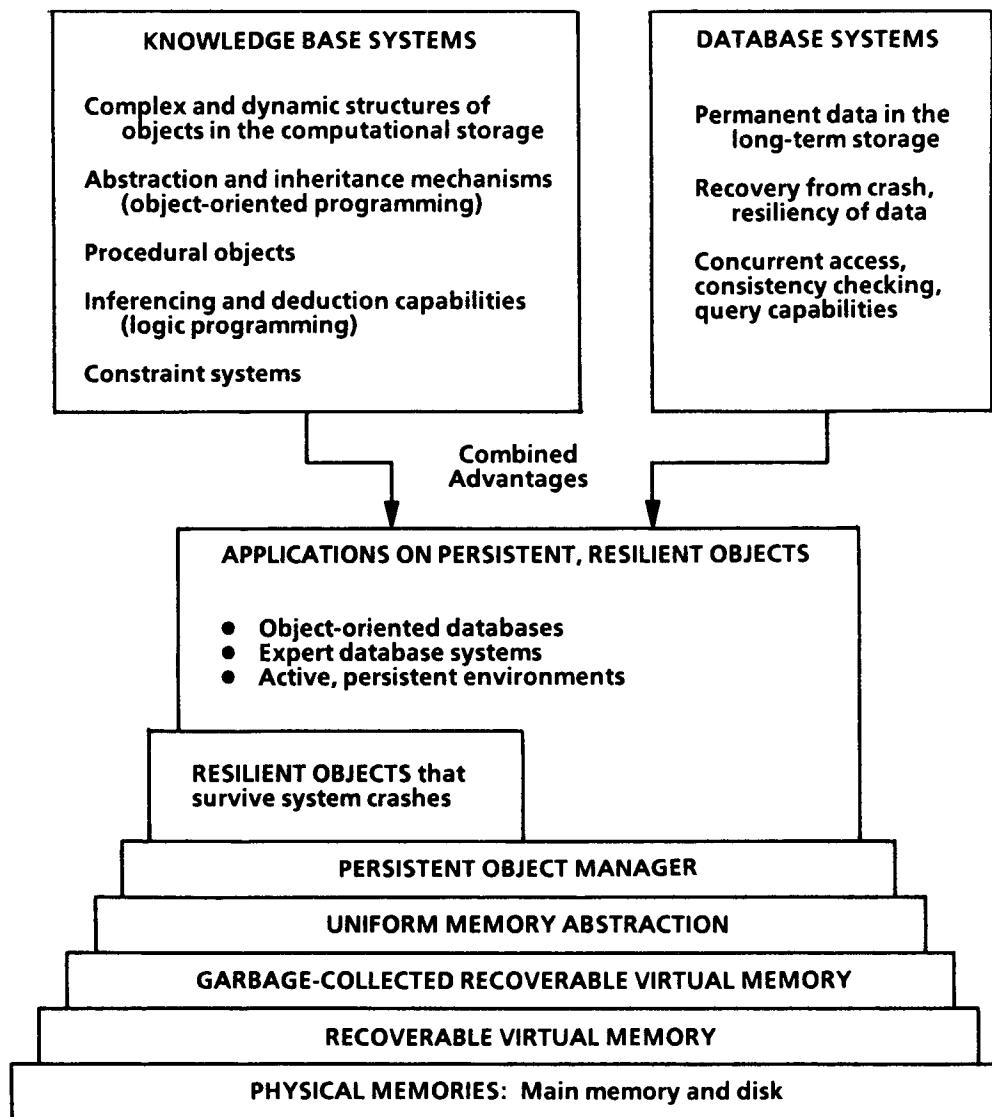


Figure 1: Our approach to object-oriented databases

by pointers. Application-level objects are constructed by combining language-level objects.

The abstraction has the notion of *persistent root*, which is a distinguished object located at a fixed virtual address and disk location. All objects that are in the *transitive closure* of the persistent root, i.e., reachable from the persistent root by following pointers, are persistent. The persistent root survives system shutdowns or crashes. Typically, the persistent root may contain a pointer to a table that points to other tables or structures of persistent objects and so on. Thus, the persistent root anchors all persistent objects. Its role is similar to the root of a directory hierarchy in a file system.

The persistence attribute of an object depends solely on whether that object can be prevented from being garbage collected even after the program that created it has terminated; this can be easily arranged by making that object a

member of the set of objects in the transitive closure of the persistent root. Persistence based solely on the persistent root rather than the properties of the storage medium allows a complete separation of the persistence attribute of an object from its type or relationship with other objects. Numbers, characters, lists, procedures, environments, etc., can be persistent objects while they exist in virtual memory. Therefore, an invocation of a procedure as a persistent object is as easy and efficient as its invocation as a transient object.

The processor contains a number of "registers."³ The processor can access a memory object, i.e., read and write its individual words, if any of its registers holds a pointer to

³The word *register* is used in a generic sense; it may be a hardware register or a scratch-pad memory in the processor.

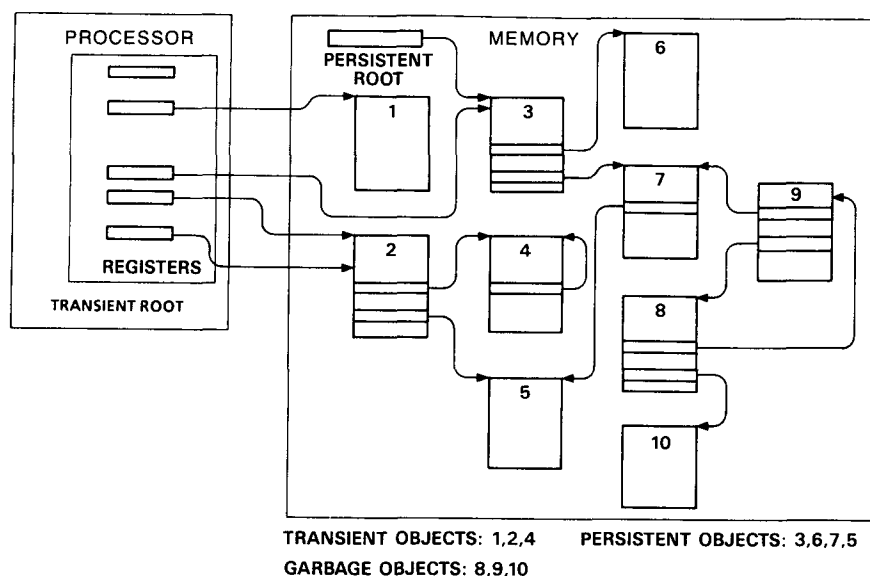


Figure 2: Uniform memory abstraction

the object. The processor can access memory objects only via *logical* addresses; a logical address consists of a pair (i, j) , where “ i ” is the id of a processor register, and “ j ” indicates the j -th word of an object being pointed at by processor register “ i .” Each memory reference can be checked for bounds, i.e., “ j ” in a logical address (i, j) should not exceed the size of the object pointed to by processor register “ i .” Registers in the processor define the *transient root* of the memory system. They do not survive a system shutdown or crash. All objects that are in the transitive closure of the transient root, but *not* in the transitive closure of the persistent root, are called transient. All the remaining objects are garbage and are reclaimed by a garbage collector. In Figure 2, objects 1, 2, and 4 are transient; objects 3, 5, 6 and 7 are persistent; and objects 8, 9 and 10 are garbage.

From the machine point of view, transient and persistent objects are indistinguishable. From the user point of view, there is no need to treat transient and persistent objects differently – hence, the name *uniform* memory abstraction: all the user needs to know is that to make an object persistent, he has to place it in the transitive closure of the persistent root. The pointers to objects serve as *names* that can be passed as procedure parameters, returned as procedure results, and stored in other objects as components. The uniform memory abstraction requires automatically *garbage collected* memory – a feature supported by Lisp machines. Thus, the abstraction provides an appropriate storage model to store both transient and persistent objects.

The persistent object manager is implemented on the uniform memory abstraction. The interface between the persistent object manager and the uniform memory abstraction defines *how* the persistent root should be accessed, i.e., both read and modified. A well-controlled *procedural interface* to the persistent root is desired instead of an interface that allows direct manipulation of the persistent root.

because accidental or malicious destruction of the persistent root can destroy the entire persistent memory which may be used as a repository that can be shared among several processes and applications. Providing application routines with direct access to the persistent root would make the interface at too low a level, since applications would be in contention for use of the root and there would be no way to keep track of persistent objects, which are often not related or user readable.

The integrity of the structure of the memory system is essential to guarantee that a roll-back process following a system crash to the last checkpointed state would be able to recover the system. Otherwise, the checkpointing operation may capture a state whose integrity has already been violated, and no recovery would be possible by rolling back the system to the corrupted checkpointed state following a system crash. The notion of the integrity of memory system refers to the integrity of the *structure* of the graph of memory objects interconnected by pointers.

The key features of the uniform memory abstraction, which forms the foundation for the integrity of the memory system, are discussed below:

1. Memory accesses only via logical addresses.
2. Unforgeability of pointers.
3. Bounds checking.
4. Automatic garbage collection
5. No run-away memory allocation

It is assumed that most user software will be written in a high-level language (such as Lisp) and not in an assembly language. The integrity of the memory system is preserved, because using a high-level language it is impossible to forge a pointer, no information outside the bounds of an object can be accessed, and objects are accessed only

via logical addresses. Therefore, the graph of memory objects undergoes transition only from one consistent state to another, ensuring its structural integrity (unless there is an undetected hardware failure that violates the integrity).

In addition, the integrity of the memory system requires that it should be automatically garbage collected, and there is no “run-away” memory allocation problem. Automatic garbage collection is essential to be able to make computational progress in a finite amount of memory space. Without the reclamation and reuse of memory space occupied by an object proven to be garbage (i.e., no outstanding pointers to the object from non-garbage objects), the system would eventually come to a halt as it would run out of memory, and at this point it could be restarted if and only if more memory resource were made available via garbage collection. Similarly, a runaway process that requests memory beyond its assigned quota must be halted and reset; otherwise, the system would eventually come to a halt as it would run out of memory.

5 Recoverable Virtual Memory

The uniform memory abstraction is implemented on top of the garbage-collected virtual memory system of a host computer. In the absence of system crashes and shutdowns, it is a straightforward matter to implement the abstraction. However, in real systems both events do occur, and sometimes rather too frequently. It must be emphasized that the problem will not disappear by making the processor’s hardware and memory free of failures because software failures may still crash the system, and crashes caused by software failures are far more common than hardware failures. In today’s computers, the storage dichotomy comes to the “rescue”; after a system crash, the contents of virtual memory are assumed to be lost, and the system is booted from a file system or a boot device. All the persistent information is usually safe in a file system and survives system crashes.

Therefore, the most challenging problem in implementing the storage architecture of persistent memory is to develop a crash recovery scheme to maintain object consistency in the presence of system crashes. In fact, at least one project (Intel iMAX-432 object filing system) decided to live with the storage dichotomy due to a lack of crash recovery schemes [Pollack81]. Recovery becomes even more challenging in memory systems of symbolic computers: a page may contain multiple objects, and an object may span multiple pages. An object can point to any other object, and other objects can hold pointers to it. Writing a single object to stable storage cannot maintain object consistency with respect to other objects, unless the entire system state is captured on disk.

Failures dealt with by a recovery scheme can be classified as *system crashes* and *disk crashes*. A system crash can occur due to power failure, hardware failure, or software error. It is signalled by a power failure interrupt, or hardware checking circuits, or software error handling routines when they cannot handle a software error or an exception condition. The rate of system crashes in a single-user ma-

chine is expected to be a few crashes per month, and the recovery time of several minutes is acceptable, assuming no permanent hardware failures. System crashes due to software errors are expected to be much more common than those due to hardware failures. The recovery scheme for system crashes is described first, followed by the treatment of disk crashes.

Our recovery scheme is inspired from the study of recovery schemes known in the conventional database community [Lorie77], [Reuter80]. However, the key difference between our recovery scheme and database recovery schemes is that our scheme is at the level of virtual memory itself. To our knowledge no existing computers have a recovery capability at this level. Our recovery scheme is based on an efficient checkpointing technique that captures the entire system state and stores it on disk in an *incremental* fashion. Changes in the memory system following the last checkpoint are incrementally accumulated on disk in *sibling* pages. The correct sibling to be fetched on a page fault and the disk block on which it is to be written are identified by means of timestamps. The scheme keeps the entire machine state valid within the last few minutes on disk. After a system crash, recovery is achieved by *rolling back* the system state to the last checkpointed state. The recovery scheme is *application-independent* and *user-transparent*.

Our recovery scheme is quite different from the Disk-save or Sysout operations on today’s Lisp machines. The disk-save operation is used to save the entire virtual address space of a TI Explorer Lisp machine into a disk band [TI85e] by *copying* the contents of the entire address space; similarly the Sysout operation is used to copy the entire virtual address space of a Xerox 1100 Lisp machine into a file [Xerox83]. Consequently, these operations are very slow (10 to 15 minutes) on today’s moderate size address spaces of about 100 M bytes. The primary use of these operations is to create a customized Lisp world by loading Lisp functions from a file system and then saving the Lisp world using the disk-save or sysout operation. The user can boot the machine using such a Lisp world. These operations are not adequate to construct a recoverable virtual memory in the sense presented in this paper. Moreover, these operations cannot be scaled up for the future bigger address spaces of several gigabytes due to unacceptable time and space overheads. In contrast, our recovery scheme is based on an efficient checkpointing operation that captures the state of machine in few seconds in an incremental fashion with modest disk space overhead as explained below. For lack of space, the paper presents only the high-level salient features of the recovery scheme. Details can be found in [Thatte85].

Page and timestamp management: A virtual page is materialized on disk in either *sibling* or *singleton* form. In sibling form, two disk blocks are allocated to a virtual page. These sibling blocks are only logical siblings and need not be physically adjacent on disk. In singleton form, a single disk block is allocated. A page is materialized in sibling form if it is expected to contain data that is likely to be modified. To reduce the disk space requirement, a page is

materialized in singleton form if it is unlikely to be modified in the future (for example, a page containing instructions). Pages in sibling form may be converted to singleton form, whenever appropriate, to save disk space as explained later in this section. However, as in a conventional virtual memory system, a virtual page occupies only a single page frame when resident in main memory.

When a page is written to disk, a timestamp that records the time of the disk write operation is generated. The page header may be used to record the timestamp. Alternatively, the page table that maps virtual addresses to disk addresses may be used to record the timestamps. Timestamps are derived from a timer that runs reliably even in the presence of system shutdowns and crashes. The granularity of timestamps need only be moderately smaller than the time for a disk write operation because pages cannot be written faster than the disk write speed. With a 10 milliseconds granularity, a 64-bit timer can generate unique timestamps for over 5.8 billion years! Therefore, a 64-bit wide field for timestamps is more than adequate.

When a page is materialized in sibling form, its siblings are assigned initial timestamps of -1 and -2, indicating that both are yet to be written.⁴ When a page is materialized in singleton form, it is assigned a timestamp of -1. All disk blocks that are modified since their initial materialization on disk will have unique timestamps within a machine.

The siblings are denoted as x and x' . $TS(x)$ and $TS(x')$ denote the timestamps of x and x' , respectively. As will soon become clear, siblings x and x' may exchange their roles when they are written to disk. A singleton page is denoted as s and its timestamp as $TS(s)$. The time of the last checkpoint operation is denoted as T_{chk} . It is stored in a reliable fashion at a known disk location.

For a singleton page s , if $TS(s) < T_{chk}$, then s belongs to the checkpointed state; if $T_{chk} < TS(s)$, s is outside the checkpointed state. For sibling pages x and x' , if $TS(x) < TS(x') < T_{chk}$ or $TS(x') < TS(x) < T_{chk}$, the sibling with the smaller timestamp contains outdated information, and the sibling with the larger timestamp belongs to the checkpointed state; if $TS(x) < T_{chk} < TS(x')$ or $TS(x') < T_{chk} < TS(x)$, the sibling with the smaller timestamp belongs to the checkpointed state, and the sibling with the larger timestamp is outside the checkpointed state. Because of the way the timestamps are initialized and updated, " $T_{chk} < TS(x) < TS(x')$ " or " $T_{chk} < TS(x') < TS(x)$ " case is not possible.

Four cases arise on a page fault depending on whether the page is in sibling or singleton form and its timestamp.

Case 1. Page fault on a sibling page, and $TS(x) < TS(x') < T_{chk}$ or $TS(x') < TS(x) < T_{chk}$: " $TS(x) < TS(x') < T_{chk}$ " case is described here. The treatment of " $TS(x') < TS(x) < T_{chk}$ " is analogous. The sibling with the larger timestamp, x' , is kept in main memory, and the other sibling, x , is discarded. When the page is written to disk, it is written over the disk space of the discarded sibling x , because x contains useless information.

⁴This initialization scheme is not unique. Other schemes are possible.

Disk space of x' must not be written over because it would destroy the checkpointed state. The timestamp relationship now becomes $TS(x') < T_{chk} < TS(x)$, i.e., case 2 below. Thus, x and x' exchange their roles.

Case 2. Page fault on a sibling page, and $TS(x) < T_{chk} < TS(x')$ or $TS(x') < T_{chk} < TS(x)$: " $TS(x) < T_{chk} < TS(x')$ " case is described here. The treatment of " $TS(x') < T_{chk} < TS(x)$ " is analogous. The sibling with the larger timestamp, x' , is kept in main memory, and the other sibling, x , is discarded. Unlike case 1, however, the page is written over its own disk space, i.e., over disk space of x' , because x' is *not* part of the last checkpointed state and can be written over, while disk space of x belongs to the checkpointed state and must not be destroyed. The timestamp relationship remains $TS(x) < T_{chk} < TS(x')$, i.e., case 2.

Case 3. Page fault on a singleton page and $TS(s) < T_{chk}$: If the singleton page is modified, at page-out time it must be converted to a sibling form because the checkpointed state must not be overwritten. Sibling x retains the contents and timestamp of the original singleton, and sibling x' contains the modified contents and the timestamp of page-out time. The timestamp relationship becomes $TS(x) < T_{chk} < TS(x')$, i.e., case 2. The disk space for s is reclaimed.

Case 4. Page fault on a singleton page and $T_{chk} < TS(s)$: At page out time, no conversion to sibling form is needed because singleton s does not belong to the checkpointed state and can be written over its own disk space. The timestamp relationship remains $T_{chk} < TS(s)$, i.e., case 4.

Sibling to singleton conversion: To reduce the disk space requirement, a sibling page may be converted back to singleton form when both siblings remain inactive for a long period, defined by a *threshold* parameter. The disk space manager hunts for such inactive sibling pages; if $TS(x) < TS(x') < T_{chk}$ and $T_{chk} - TS(x') < \text{threshold}$, then the disk space for both siblings x and x' is reclaimed by converting them into singleton form. Singleton s contains the original contents and timestamp of sibling x' . The treatment for " $TS(x') < TS(x) < T_{chk}$ and $T_{chk} - TS(x) < \text{threshold}$ " is analogous.

Checkpoint process: The checkpoint process may be initiated by an application or the system. At checkpoint time, the checkpoint process saves all processor registers, i.e., the transient root into a *snapshot* object. The snapshot object is part of virtual memory. The page containing the snapshot object is written to disk. The snapshot object may be incorporated in the transitive closure of the persistent root to make it persistent. All dirty pages in main memory are then written to disk. Finally, T_{chk} is updated on disk to the current time, completing the checkpoint process. This update operation must be implemented as an *atomic* operation; T_{chk} is either successfully updated or it does not change at all. Right after the checkpoint completion, for all sibling pages, $TS(x) < TS(x') < T_{chk}$ or $TS(x') < TS(x) < T_{chk}$, and for all singleton pages, $TS(s) < T_{chk}$.

Post-crash recovery: It is assumed that after a system crash diagnostics are run to detect permanent hardware failures and faulty hardware, if any, is already replaced. Figure 3 indicates the post-crash recovery process described below.

1. Rollback the system to the last checkpointed state by restoring the processor registers from the snapshot object on disk.
2. Reconcile with the external world. Part of the restored checkpointed state is related to the system configuration and I/O interfaces specific to the checkpoint time. This state must now be reconciled with the current time and configuration.⁵
3. Resume the normal system operation.

Disk crashes: Disk crashes arise from disk head crashes, deterioration of the magnetic media itself, or bugs in the disk driver software. On a single-user workstation, the disk crash rate is expected to be a few failures per year, with the recovery time of a few hours. Disk crashes are treated differently from system crashes because they may corrupt the checkpointed state on disk; therefore, the roll-back technique used for recovery from system crash may not work. To deal with disk crashes, the last checkpointed state on disk needs to be archived on another media, such as a streaming tape. This operation is expected to be performed a few times a week, preferably as an overnight operation – a scenario consistent with the expected disk crash rate of few failures per year, with the recovery time of a few hours. After a disk crash, a failed disk needs to be replaced with a new one, which is then initialized from the last archived checkpointed state.

6 Resilient Objects

If checkpoints are taken frequently, say every ten minutes, the user may lose at most the last ten minutes of work

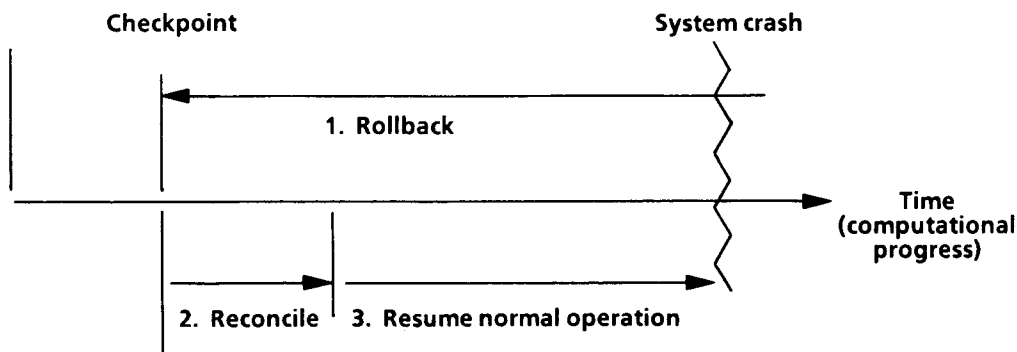


Figure 3: Post-crash recovery

due to a system crash. This may be quite acceptable for interactive program development and many applications on a personal machine. However, for some applications, such as graphics or text editing, loss of work may be an irritating annoyance, if not a calamity. What these applications need is object resilience.

A persistent memory system can support resilient objects with the help of a transaction⁶ management package. For the sake of concreteness, it is assumed that the transaction package is implemented using *undo* and *redo* logs to advance the state of the machine beyond the last checkpoint. As an alternative scheme, a transaction package can be implemented by writing objects changed by a committed transaction to a stable storage. When a client process initiates a transaction, the transaction manager executes it by following a protocol similar to the two-phase locking protocol [Eswaran76] to allow concurrent transactions. A transaction management system implemented directly on top of persistent memory is briefly described below. The undo and redo logs are persistent objects.

1. Acquire all locks on the required objects from the lock manager.
2. For every *write* operation, create appropriate entries in the undo and redo logs for the transaction. Apply consistency checks to decide whether the transaction should be committed or aborted. If it is to be committed, go to step 3. If it is to be aborted, use the undo log to undo changes; release all locks; discard the undo and redo logs, and exit.
3. Write information in the redo log in persistent memory to the *external* redo log kept on disk *outside* virtual memory.⁷ The redo log in persistent memory may now be discarded.
4. Notify the client process of successful completion of the transaction, and release all locks acquired in step 1. The undo log in persistent memory may now be discarded.

⁵For example, part of the restored checkpointed state may contain a timer variable, which reflects the time of the last checkpoint. The timer variable must be adjusted to reflect the current time. The state of I/O device registers and device control blocks may need to be adjusted so that the I/O devices become ready for normal operation.

⁶The notion of transactions is due to Eswaran et. al. [Eswaran76].

⁷This operation is necessary for the survival of the redo information of committed transactions from system crashes. The external redo log survives system crashes because it is kept *outside* virtual memory.

The post-crash recovery process for resilient objects consists of the following additional steps after step 2 of Figure 3. These steps re-establish applications to a state consistent with committed transactions.

- Apply the undo log found in the checkpointed state in the *reverse time order*. If a checkpointed state contains changes made by an uncommitted transaction, it will also contain the corresponding undo log entries required to undo the changes.
- Apply the external redo log maintained outside virtual memory in the *forward time order*. Now the normal system operation can be resumed.

Database systems can be constructed as applications on top of resilient objects. Since the persistent memory can support complex object types and structures, resilient objects also enjoy the same benefits. Therefore, a database object can contain pointers to arbitrary objects, such as procedures, lists, and other arbitrary structures. Complex structures of objects including *cyclical* structures, which are quite common in symbolic computing, can be supported.

Our approach of constructing a database system on top of resilient objects has great flexibility and representational power, and it should be contrasted with the current approach of implementing resilient objects on top of a conventional file system or database. The current approach is quite rigid; it can support neither arbitrary types of objects nor complex structures. In addition, our approach is expected to be easier to implement than conventional database systems: No file or database buffers or explicit I/O operations are needed. If the transaction commits before a crash, then as presented above, its redo information is already written to the external redo log and the undo information is discarded. Therefore, there is no need to maintain an external undo log.

7 Persistent Object Manager

As shown in Figure 4, the persistent object manager provides client application programs with an interface to persistent memory. The client interface provides two main services to client applications: the first service is a *namespace service* which supports operations for creating and managing a namespace of mappings of user-supplied names into Lisp objects; and the second service provides the undo and redo log primitives for implementing atomic transactions required for resilient objects, as described in Section 6. The persistent object manager does *not* itself implement any existing database data model, such as the relational data model; instead, it provides one step above the persistent memory on which to interface to application programs directly or to support higher level data managers.

It is desirable to provide a namespace facility in the persistent object manager. The idea of providing a namespace facility is not new in itself. The Xerox Clearinghouse [Oppen81] provides a namespace for naming and locating various objects, such as machines, workstations, file users, and people in a distributed environment. The namespace facility planned within the persistent object manager provides a mapping of user-supplied names into persistent objects within a single machine. What is new is that the namespace names and their related objects are themselves persistent since they are accessible from the system's persistent root. Namespace names are intended to serve as application-specific persistent roots available to applications as needed and providing access paths to persistent Lisp objects. Deleting a name-object association from a namespace deletes an access path to an object and does not necessarily affect the object itself.

Names in a namespace provide convenient hooks for associating with an object application-specific data type information, import-export functions for archiving or re-

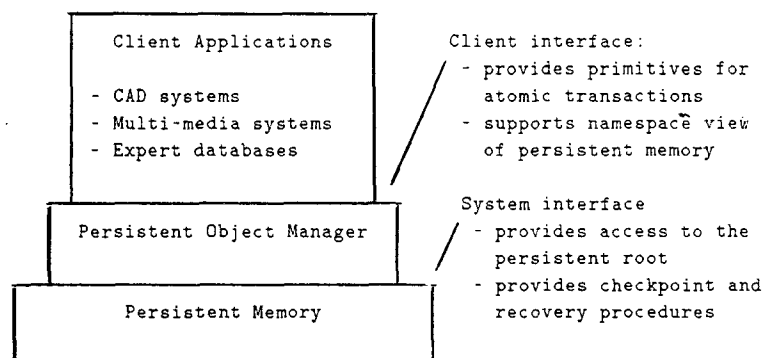


Figure 4: Persistent object manager and its interfaces

trieving an object in the file system, archiving reasons, protection and locking information, usage statistics (date last used, “by whom” information, etc.), version or time-stamp information, and human- or machine-readable documentation. Namespace names divide the persistent space into shared or disjoint persistent spaces. An access control scheme (using passwords, for instance) can be built that gives users access to only part of the namespace, making it possible to guarantee that transitive closures of some roots are disjoint from transitive closures of others.

8 Persistent Memory Prototype on Explorer

We are in the process of implementing a persistent memory prototype using the TI Explorer Lisp machine. The goals of the prototype are to demonstrate the persistent memory and a persistent object manager in symbolic computing environment, investigate the interactions among the virtual memory management, recovery, and garbage collection, evaluate the robustness of the recovery scheme, and understand the utility of persistent memory for some real applications, such as a VLSI CAD system. An Explorer supports up to 128 M bytes of virtual address space [TI85a]. Since the prototype will be limited by the 128 M bytes of address space, it will not be able to store a large amount (exceeding 128 M bytes) of persistent object information. However, we believe that in spite of the address space limitations, the prototype will help obtain significant insights, which come only with a real implementation.

Disk bands for virtual memory on Explorer are organized as 16-page *clusters*. A cluster consists of 16 consecutive virtual pages. The virtual memory manager uses the Disk Page Map Table (DPMT) – a memory-resident data structure – to map a virtual address to disk address after a page fault. The DPMT contains mapping information for the clusters. For the persistent memory prototype, DPMT is modified to keep time-stamp related information. On a page fault, this information is used to decide which sibling to fetch from the paging band on disk. Siblings are only *logical* siblings, i.e., they need not be physically adjacent on disk.

It is highly desirable to reduce the time required for the checkpoint operation so that the user processes are not suspended too long. This is achieved by keeping the fraction of dirty pages in main memory at the checkpoint time reasonably small with a background write process that continually cleans up dirty pages between successive checkpoints. Normally this process is scheduled as a low-priority process so that user processes are not adversely affected. When the checkpoint operation is initiated, the priority of this process is increased to complete the cleaning of all the remaining dirty pages without any preemption by the scheduler.

Dirty page clean-up requires disk bandwidth in addition to that needed for the normal paging activity. Our calculations show that for typical processors (speed: 1 to 4 million instructions per second), typical disk systems (access time:

30 to 50 milliseconds), and a moderate page fault rate (page fault period: 500K to 1 M machine instructions per page fault), 50% to 80% disk bandwidth is available to clean-up dirty pages, and convert siblings to singletons and singletons to siblings. This bandwidth is judged to be adequate to support a recoverable virtual memory on an Explorer. The scatter-gather feature of the Explorer disk system is used to write out dirty pages in as few disk accesses as possible. With a vigorous background write process and use of the scatter-gather capability, we expect that the checkpointing operation should be over in few seconds.

For the persistent memory prototype, the garbage collector also needs to be modified; the garbage collector reclaims the disk space for a cluster if all its 16 pages contain garbage. In the persistent memory, the underlying disk space for a garbage cluster cannot be immediately reclaimed if any of its virtual page belongs to the checkpointed state. If the disk space were to be reclaimed, it will disturb the checkpointed state. For the persistent memory, reclamation of the disk space for a garbage cluster is postponed until the next checkpoint: after the completion of the next checkpoint, the disk space corresponding to the garbage clusters does not belong to the checkpointed state, and can be safely reclaimed. Note that it is the *disk space* of garbage clusters whose reclamation needs to be postponed, and not the *virtual pages* of garbage clusters. The virtual pages can be reclaimed and made available to the memory allocator immediately.

9 Future Work

We are investigating the extensions of persistent memory in a network-based distributed system. The motivation of this effort is to be able to share objects in a distributed environment. Similar research efforts are underway at other manufacturers of Lisp machines; the MOBY address space project at the Lisp Machine Inc. is an example of such efforts [Greenblatt85].

The distributed computing environment envisioned consists of several symbolic workstations connected among themselves and with one or more file servers via a local-area network. Each workstation's virtual address space is divided into *persistent areas*. An area is the unit of garbage collection like Bishop's areas [Bishop77]. It is called “persistent” area because it is also a unit of recovery, and persists across system crashes. Our approach is expected to solve the two very difficult problems in this type of computing environment, viz., network-wide garbage collection, and network-wide recovery. Since a persistent area is the unit of garbage collection as well as recovery, it can be garbage-collected independent of any other areas. Similarly, it can be independently recovered from crashes. There is neither any dependence on nor any cooperation required from other workstations to complete garbage collection or recovery of an area.

10 Conclusion

We strongly believe that the storage dichotomy in today's computers will be a serious impediment in the development of object-oriented database systems. The storage dichotomy puts a heavy burden of storage management on the programmer, and causes a space and time penalty due to the translation and transfer of information between virtual memory and databases. Our approach to object-oriented databases is based on persistent memory that eliminates the distinction between transient and persistent objects. Implementation of persistent memory requires a recovery capability at the level of virtual memory itself. Our recovery scheme is based on timestamp and sibling page techniques, and has low space and time overheads. Resilient objects can be implemented on top of a persistent memory. Many applications that include CAD systems, multi-media systems, expert database systems, and object-oriented databases are expected to be implemented with greater ease, flexibility and performance on persistent, resilient objects than on a file system or on a conventional database.

References

- [Abelson85] H. Abelson and G. J. Sussman, *Structure and Interpretation of Computer Programs*, MIT Press, Cambridge, MA, 1985.
- [Atkinson83] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, P. W. Cockshott, and R. Morrison, "An approach to persistent programming," *The Computer Journal*, vol. 26, no. 4, pp. 360-365, December 1983.
- [Atkinson84] M. P. Atkinson, P. Bailey, W. P. Cockshott, K. J. Chisholm, and R. Morrison, "Progress with persistent programming," Technical Report RPR-8-84, Department of Computer Science, University of Edinburgh, Edinburgh, Scotland, February 1984.
- [Bensoussan69] A. Bensoussan, C. T. Clingen, and R. C. Daley, "The MULTICS virtual memory," In *Proc. 2nd Symp. Operating Systems Principles*, pp. 30-42, Princeton University, October 1969.
- [Bishop77] P. B. Bishop, "Computer systems with a very large address space and garbage collection," Technical Report TR-178, Laboratory for Computer Science, Cambridge, MA, May 1977.
- [Butler86] M. H. Butler, "An approach to persistent LISP objects," In *Proc. COMPCON*, pp. 324-329, IEEE, San Francisco, CA, March 1986.
- [Cockshott84] W. Cockshott, M. Atkinson, K. Chisholm, P. Bailey, and R. Morrison, "Persistent object management system," *Software Practice and Experience*, vol. 14, pp. 49-71, 1984.
- [Eswaran76] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, "The notions of consistency and predicate locks in a database system," *CACM*, vol. 19, no. 11, pp. 624-633, November 1976.
- [Greenblatt85] R. Greenblatt, "MOBY address space," Lisp Machine Inc., Los Angeles, CA, August 1985, Seminar report on research in progress.
- [Hartzband85] D. J. Hartzband and F. J. Maryanski, "Enhancing knowledge representation in engineering databases," *IEEE Computer*, vol. , pp. 39-48, September 1985.
- [Herot80] C. F. Herot, "SDMS: Towards spatial data management system," *ACM TODS*, vol. 5, no. 4, pp. 493-514, April 1980.
- [IBM38] IBM General Systems Division, "IBM System/38 technical developments," Technical Report G580-0237-1, IBM, July 1980.
- [Kilburn62] T. Kilburn, "One-level storage system," *IRE Trans. Electronic Comput.*, vol. EC-11, no. 2, , April 1962.
- [Lorie77] R. A. Lorie, "Physical integrity in large segmented database," *ACM TODS*, vol. 2, no. 1, pp. 91-104, March 1977.
- [McDonald81] N. McDonald and P. J. McNally, "VGQF: Video Graphics Query Facility database design," In *Proc. ACM SIGMOD SIGSMALL Workshop*, pp. 96-101, 1981.
- [McEntee86] T. J. McEntee, "An overview of garbage collection in symbolic computing," *Texas Instruments Engineering Journal*, vol. 3, no. 1, pp. 130-139, January 1986.
- [McKewon84] D. M. McKewon, "Digital cartography and photo interpretation from a database viewpoint," In G. Gardarin and E. Gelenbe, editors, *New applications for data bases*, pp. 19-42, Academic Press, 1984.
- [Mishkin84] N. Mishkin, "Managing permanent objects," Technical Report YALEU/DCS-RR-338, Department of Computer Science, Yale University, New Haven, CT, November 1984.
- [Moon84] D. A. Moon, "Garbage collection in a large Lisp system," In *Proc. 1984 ACM Symp. Lisp and Functional Programming*, pp. 235-246, August 1984.
- [OOS85] Edited by F. Lochovsky, "IEEE Database Engineering," December 1985. A quarterly bulletin of the IEEE Computer Society Technical Committee on Database Engineering, Special Issue on Object-Oriented Systems.

- [Oppen81] D. C. Oppen and Y. K. Dalal. "The Clearinghouse: A decentralized agent for locating named objects in a distributed environment." Technical Report OPD-T8103, Systems Development Department, Xerox Corporation, Palo Alto, CA, October 1981.
- [Pollack81] F. J. Pollack, K. C. Kahn, and R. M. Wilkinson, "The iMAX-432 object filing system," In *Proc. 8th ACM Symp. Operating Systems Principles - ACM SIGOPS Operating Systems Review*, pp. 137-147, ACM SIGOPS, Pacific Grove, CA, December 1981.
- [Reuter80] A. Reuter, "A fast transaction-oriented logging scheme for UNDO recovery," *IEEE Trans. Software Eng.*, vol. SE-6, no. 4, pp. 348-356, July 1980.
- [Saltzer78] J. H. Saltzer, "Naming and binding of objects," In R. Bayer, R. M. Graham, and G. Seegmuller, editors, *Operating Systems: An Advanced Course*, p. 99, Springer-Verlag, New York, NY, 1978.
- [Stonebraker84] M. Stonebraker and A. Guttman. "Using a relational database management system for computer aided design data - an update," *IEEE Database Engineering*, vol. 7, no. 2, pp. 56-60, June 1984.
- [TI85a] Texas Instruments Incorporated, Data Systems Group, "EXPLORER Technical Summary," Austin, TX, May 1985, Part No. 2243189-0001.
- [TI85e] Texas Instruments Incorporated, Data Systems Group, "EXPLORER Programming Concepts and Tools," Austin, TX, May 1985, Part No. 2243130-0001.
- [Thatte85] S. M. Thatte. "Persistent memory for symbolic computers," Technical Report TR-08-85-21, Central Research Laboratories, Texas Instruments Incorporated, Dallas, TX, July 1985.
- [Thatte86] S. M. Thatte, "Persistent Memory: merging AI-knowledge and databases," *Texas Instruments Engineering Journal*, vol. 3, no. 1, pp. 151-159, January 1986.
- [Traiger82] I. L. Traiger. "Virtual memory management for database systems," *ACM Operating System Review*, vol. 16, pp. 26-48, October 1982.
- [Xerox83] Xerox Corporation, "Xerox Interlisp Reference Manual," October 1983.