Cricket: A Mapped, Persistent Object Store

by
Eugene Shekita
Michael Zwilling

Computer Sciences Technical Report #956
August 1990

Cricket: A Mapped, Persistent Object Store

Eugene Shekita
Michael Zwilling

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

This research was partially supported by DARPA under contracts N00014-88-K-0303 and NAG 2-618, NSF under grant IRI-8657323, and donations from Texas Instruments and Digital Equipment Corporation.

Cricket: A Mapped, Persistent Object Store

Eugene Shekita
Michael Zwilling

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

ABSTRACT

This paper describes Cricket, a new database storage system that is intended to be used as a platform for design environments and persistent programming languages. Cricket uses the memory management primitives of the Mach operating system to provide the abstraction of a shared, transactional single-level store that can be directly accessed by user applications. In this paper, we present the design and motivation for Cricket. We also present some initial performance results which show that, for its intended applications, Cricket can provide better performance than a general-purpose database storage system.

(To appear in *Proc. of the 4th Intl. Workshop on Persistent Object Systems Design, Implementation and Use*)

1. INTRODUCTION

In recent years, there has been a great deal of research in extending database technology to meet the needs of emerging database applications such as text management and multi-media office systems (see [DBE87] for a good survey). Out of this research has come a variety of new storage systems that attempt to provide more functionality as well as improved performance for these emerging applications. Examples of such systems include [Care86, Horn87, Lind87, Moss88, Sche90, Ston90]. While these storage systems will undoubtedly meet the performance demands of many new applications, our view is that for some applications there is still considerable room for improvement. In particular, we feel that for design environments [Katz87, Chan89], persistent programming languages [Cock84, Atki87], and other applications in which response time rather than throughput is often the key concern, different storage techniques that those currently in use can provide better performance.

Towards this goal, we have designed a new database storage system called Cricket.[1] Cricket uses the memory management primitives of Mach [Acce86] to provide the abstraction of a shared, transactional, single-level store. One advantage of a single-level store is that it provides applications with a uniform view of volatile and non-volatile (i.e., persistent) memory. This in turn can lead to improved performance by eliminating the need for applications to distinguish and convert between non-persistent and persistent data formats [Cope90].

---

[1] As the reader shall see, the flow of control really hops around in our storage system!

Although storage systems based on a single-level store have been proposed as far back as Multics [Bens72], Cricket offers several features that have not been combined in one system before. One of Cricket's key features is the ability to let applications directly access persistent data, but at the same time maintain the applications in separate (and potentially distributed) protection domains. Cricket also offers transparent concurrency control and recovery, and since it runs as a user-level process on Mach, it is easily ported to a variety of machines. We believe that these features distinguish Cricket from other recent proposals based on a single-level store [Chan88, Ford88, Spec88, Cope90] and make it an attractive platform for design environments and persistent languages.

The remainder of this paper provides a detailed description of Cricket. In the next section, we present the motivation for Cricket, and then in Section 3, we argue why a single-level store is the right approach. This is followed by Section 4, where we review Mach's external pager facilities [Youn87], which play a central role in Cricket's design. Cricket's system architecture is then described in Section 5, and in Section 6, we provide some preliminary performance results that compare Cricket to the EXODUS Storage Manager [Care86]. These preliminary results show that for its intended applications, Cricket can provide better performance than a general-purpose database storage system. Finally, related work is mentioned in Section 7, and conclusions are drawn in Section 8.

2. THE MOTIVATION BEHIND CRICKET

While traditional database storage systems are extremely good at retrieving large groups of related objects and performing the same operation on each object, they are generally ill-suited for design environments. To illustrate why, it is useful to step through the execution of a design transaction in a CAD/CAM system [Katz87]. There, transactions can be broken down into three basic phases: 1) a loading phase, when the design is loaded into memory from disk, 2) a work phase, during which the design is repeatedly changed, and 3) a saving phase, when design changes are committed. As noted in [Maie89], this load/work/save paradigm is substantially different from a traditional database workload. During the work phase, accesses are unpredictable and fast response time is the key performance criteria rather than system throughput. Moreover, the data objects that make up the design may be traversed and updated hundreds, even thousands of times before the design is saved.

Unfortunately, traditional database storage systems are not geared for these sort of access patterns. Among other things, the procedure-based interface that must typically be used to traverse and update persistent objects is too slow [Moss90]. And as noted in [Maie89], the recovery protocols are often inappropriate. For example, generating a log record for each update in the work phase of a design transaction would obviously have a negative impact on response time (not to mention the volumes of log data that could be generated). For these reasons, CAD transactions often use a database system in more of a batch mode by loading a whole design into their virtual address space, converting it to an in-memory format, working on it, converting it back to a disk format, and then

committing the entire design as changed at end-of-transaction. In general, we would argue that these problems are not just limited to design environments. Implementors of persistent languages have already run up against many of the same problems [Rich89, Schu90].

More recently, a number of new database storage systems have been proposed to address some of these issues, e.g., [Care86, Horn87, Lind87, Moss88, Sche90, Ston90]. But our feeling is that for design environments and persistent languages, many of these systems will still fall short of the mark. This is due to the fact that many of them still use a procedure-based interface to access persistent data. Moreover, many of them still use fairly traditional recovery techniques based on write-ahead logging [Moh89a]. It was these observations and also our experiences with the EXODUS Storage Manager [Care86] and the persistent language E [Rich89, Schu90] that motivated us to design Cricket.

3. THE ARGUMENT FOR A SINGLE-LEVEL STORE

As mentioned earlier, Cricket provides the abstraction of a single-level store to applications, and we advertise this as one of its key features. With a single-level store, the database itself is mapped into the virtual address space, allowing persistent data to be accessed in the same manner as non-persistent data. This is in contrast to a conventional two-level store, where access to persistent data is less direct and a user-level buffer pool is typically maintained to cache disk pages.

Single-level stores are nothing new, of course. Their origins can be traced back almost 20 years to Multics [Bens72], and many operating systems provide mapped file facilities that effectively implement a single-level store. But more importantly, database implementors have repeatedly rejected the idea of using the mapped file facilities offered by operating systems and instead have chosen to manage buffering and disk storage themselves. There are a variety of reasons given why this is so (see [Ston81, Trai82, Ston84]). Among the most notable are:

- Operating systems typically provide no control over when the data pages of a mapped file are written to disk, which makes it impossible to use recovery protocols like write-ahead logging [Moh89a] and sophisticated buffer management [Chou85].
- The virtual address space provided by mapped files, usually limited to 32 bits, is too small to represent a large database.
- Page tables associated with mapped files can become excessively large.

As pointed out in [Eppi89] and [Cope90], however, these criticisms may no longer be as valid as they once were. The above items can be countered by arguing that:

- With the right operating system hooks, it is possible to control when the data pages of a mapped file are written to disk. Mach, for example, provides many of the necessary hooks with its notion of *memory objects* [Youn87]. More will be said about this shortly.

- For many emerging database applications, a 32-bit address space is sufficient. Moreover, with the rapid increase in memory sizes and with shared-memory multiprocessors becoming more commonplace, processors with large virtual address spaces may soon become available. In fact, IBM's RS/6000 [Bako90] already supports a 52-bit address space, and HP's Precision Architecture [Maho86] supports a 64-bit address space (although strictly speaking, these are both segmented architectures).

- As the cost of memory decreases, large page tables will become less of a concern. Furthermore, inverted page tables such as those found in IBM's RS/6000 and HP's Precision Architecture may become more common with the increase in memory sizes. Inverted page tables exhibit the desirable property of growing in proportion to the size of physical memory rather than the size of virtual memory.

Despite these compelling arguments (see [Eppi89] for several more), the jury is still out on whether a single-level store offers any advantages for traditional database applications. In fact, the performance results presented in Eppinger's Ph.D thesis and also in [Duch89] seem to argue that it may not be a good idea for transaction processing. Interestingly enough, the real problem with a using single-level store for transaction processing appears to be the high cost of handling page faults for persistent data rather than the criticisms mentioned above.

3.1. Why a Single-Level Store is Right for Cricket

Given the known problems with a single-level store, why do we think it is the right choice for Cricket? The answer is that we are primarily interested in supporting non-traditional applications where, in our opinion, the advantages of using a single-level store outweigh its disadvantages. In the following paragraphs, we briefly mention some of these advantages.

One advantage is that a single-level store can eliminate the need for applications to distinguish and convert between non-persistent and persistent data formats. In most database storage systems, the format of persistent data and the access to it usually differs from that of non-persistent data. Moreover, the cost of accessing persistent data is generally more expensive, even after it has been brought into memory [Moss90, Schu90]. As a result, applications often convert persistent data to a more efficient in-memory format before operating on it. Unfortunately, this can involve copying costs, added buffering requirements, and format conversions. With a single-level store, non-persistent and persistent data can have a uniform representation and these costs can be eliminated [Cope90]. This has obvious benefits in applications like design environments, where the real-time cost of accessing and updating persistent data is a key concern.

For similar reasons, we feel that a single-level store will also simplify the job of implementing a persistent language. To reduce the cost of accessing persistent data, persistent languages often use "pointer swizzling" [Cock84, Moss90, Schu90]. In pointer swizzling, the embedded object identifiers (i.e., pointers) that are stored in persistent objects are typically converted to virtual addresses while they are in memory. This is done to reduce the cost of traversing objects. Unfortunately, swizzling is not as simple as it sounds. There are the issues of what identifiers to swizzle, when to swizzle them, and how to unswizzle them. And in persistent languages based on C [Agra89, Rich89], it is often difficult to know where identifiers are located, when they change, and when they need to be reswizzled [Schu90]. With a single-level store, object identifiers become virtual memory addresses, so all this effort (and its associated cost) can be eliminated.

Yet another advantage of using a single-level store is that persistence and type can be kept orthogonal [Atki87]. That is, application code can be written without concern for whether it is operating on non-persistent or persistent data. This simplifies code development and also allows binaries that were originally designed to operate on non-persistent data to be used with persistent data — which, of course, has obvious practical and commercial advantages.

Finally, a single-level store can simplify the management of persistent objects that span multiple disk pages. Because a single-level store makes use of MMU hardware, multi-page objects can be made to appear in memory as though they were contiguous without actually requiring physical contiguity. This is in contrast to the EXODUS Storage Manager [Care86], where considerable effort was required to implement contiguous buffering of multi-page objects.

4. EXTERNAL PAGERS IN MACH

In the next section, we will describe Cricket's system architecture. Before we can do that, however, we need to briefly go over Mach's *external pager* interface [Youn87], since it plays a central role in Cricket's design.

Among other things, Mach provides a number of facilities that allow user-level tasks (i.e., processes) to exercise control over virtual memory management. Mach provides the notion of a *memory object*, which is simply a data repository that is managed by some server (in this case Cricket). Such a server is called an *external pager*. An external pager is in charge of paging the data of a memory object to and from disk.

In Mach, tasks can associate (i.e., map) a given region of their address space to a memory object using the *vm_map* kernel call. After doing so, the external pager for that memory object will be called by the Mach kernel when a page in the mapped region needs to be read or written to disk. Physical sharing of data occurs when more than one task maps the same memory object into its address space. The Mach kernel and external pagers coordinate their activity through a message-based interface, which is summarized in Table 1.

Mach Kernel to External Pager Interface	
memory_object_data_request ( )	request for data page of a memory object
memory_object_data_write ( )	request to write page of a memory object to disk

External Pager to Mach Kernel Interface	
memory_object_data_provided ( )	supplies kernel with data page
memory_object_data_unavailable ( )	tells kernel to use zero-filled page

Table 1: External Pager Interface

Figure 1 illustrates how the Mach kernel and an external pager coordinate their activity on a page fault for a memory object M. At startup, the external pager acquires a *port* (i.e., capability) from the Mach kernel and associates it with M. Through an exchange of messages, the capability for M is passed to the client task, which then calls *vm_map* to map M into its address space. (Alternatively, the external pager can call *vm_map* on behalf of the client if it has the right permissions.) When the client attempts to access a page P in the mapped region, a page fault is generated. The page fault is caught by the Mach kernel, which verifies the client's access permissions and then sends a *memory_object_data_request* message to the external pager, asking it to supply the data for P. The external pager reads the data from disk and provides it to the kernel via *memory_object_data_provided.* The kernel then locates a free page frame, copies the data into the frame, and resumes the client. Subsequent accesses to P will not generate a page fault.

By default, Mach uses an LRU replacement algorithm to manage kernel memory. If its free-page list starts to run low and page P is at the top of the inactive list, then P will be replaced. If P is clean, its contents are simply

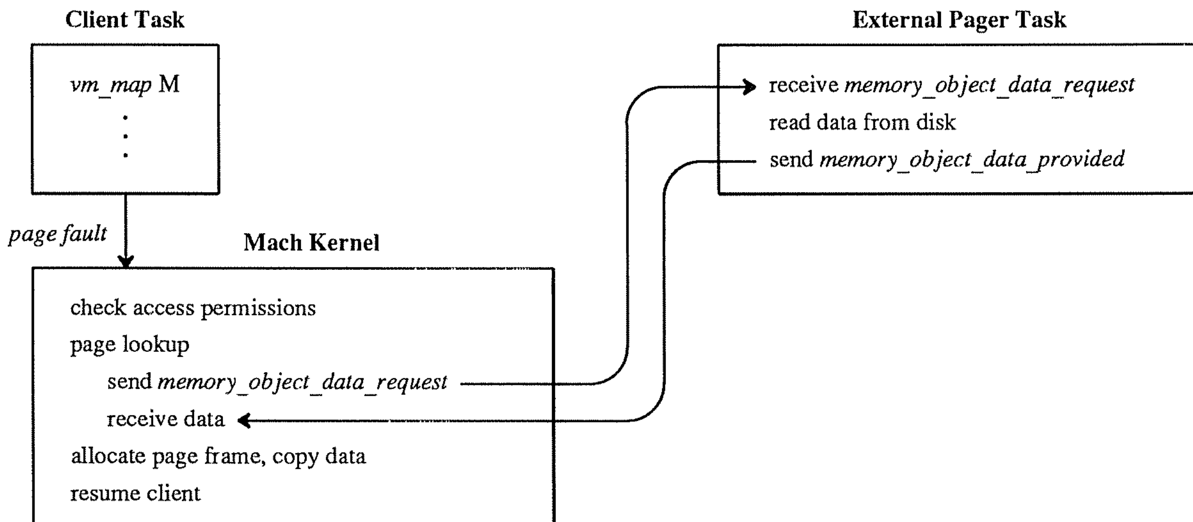

Figure 1: Handling a Page Fault on a Memory Object

discarded. Otherwise, the kernel sends a *memory_object_data_write* message to the external pager with a pointer to P, at which point the external pager is expected to write P to disk.

In addition to the interface calls mentioned above, Mach also provides the means for an external pager to force a page of a memory object to be cleaned or flushed. This effectively allows the external pager to control (to some extent) the replacement policy used for a memory object. Furthermore, *memory_object_data_provided* can be called asynchronously, so prefetching data for a memory object is also possible.

5. CRICKET'S SYSTEM ARCHITECTURE

This section describes Cricket's system architecture. The section is broken into two parts. In the first part, we discuss Cricket's basic design, and in the second part we discuss more advanced design issues that are largely unresolved at this point in time.

5.1. Basic Design

5.1.1. Architecture Overview

Figure 2 illustrates what the single-site architecture of Cricket looks like. As shown, Cricket follows a client/server paradigm. Client applications run as separate tasks, each in their own protection domain, and they use an RPC interface to request basic services from Cricket. The RPC interface includes *connect* to establish a connection with Cricket, *disconnect* to break a connection, *begin_transaction* to begin a transaction, and *end_transaction* to end a transaction. For efficiency, some of Cricket's functionality is split between the Cricket server itself and a runtime library that gets linked with the application code at compile time. The runtime library includes RPC stubs as well as code for allocating persistent data. More will be said about this shortly.

The Cricket server is multi-threaded to permit true parallelism on multiprocessors and also to improve throughput by permitting threads to run even when others are blocked on synchronous events like I/O. The Mach C-Threads package [Drav88] is used to create and manage threads. When Cricket starts up, it creates a pool of threads which all line up on the same central message queue waiting to service client or kernel requests. A given thread is not tied to any particular function or transaction. When a thread finishes servicing a request, it puts itself on the central message queue again and waits for yet another request. Mach takes care of preemptively scheduling individual threads.

As mentioned earlier, client applications are allowed direct (shared) access to persistent data. This is accomplished using Mach's external pager facility. We simply treat the database as a memory object and have the Cricket server play the role of its external pager. When a client first *connects* to Cricket, a *vm_map* call is executed by Cricket on behalf of the client to map the database into the client's virtual address space. The *connect* call returns

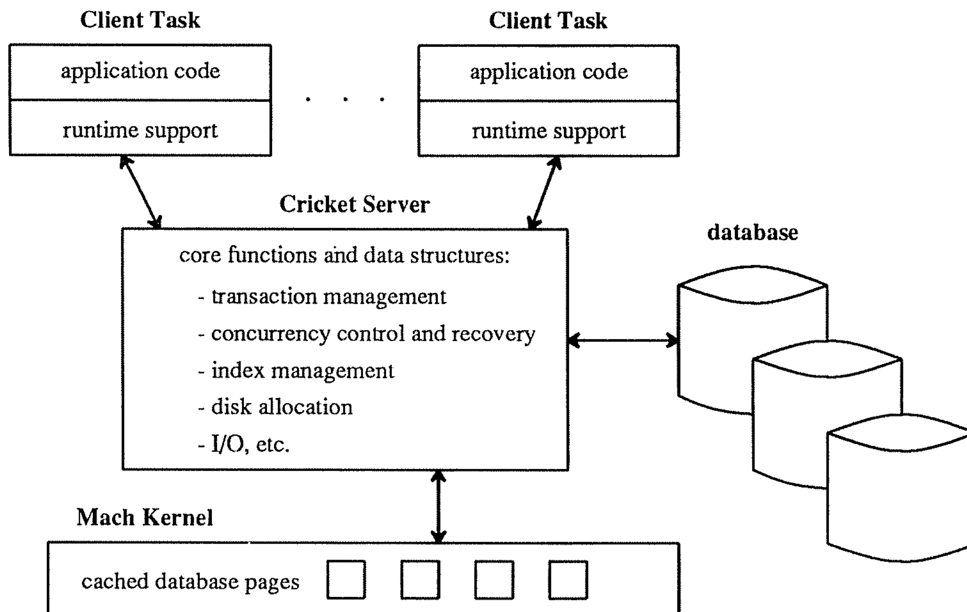

Figure 2: Single-Site Cricket

the virtual address that corresponds to the start of the database, as mapped in the client's address space. Using this address, the client can then access the database just as if it were in virtual memory — ala a single-level store. To ensure that pointers to persistent data remain valid over time, Cricket always maps the database to the same range of virtual addresses

It is important to note that database I/O is completely transparent to client applications as a result of using Mach's external pager facility. By default, the same holds true for concurrency control and recovery — although for efficiency, we are also experimenting with options that make those functions less transparent.

5.1.2. On Protection verses Performance

As shown in Figure 2, Cricket's core functions and their associated data structures are isolated in the Cricket server where they are protected from client applications. Because of its widespread use, our view is that nobody will take us seriously if we are unable to support applications written in C [Kern78] or its derivatives. Consequently, separate protection domains are a necessary evil. (One can imagine the damage a buggy C application could inflict if it had access to the disk allocation bitmaps!) In commercial database systems, the application code and the system software typically reside in separate address spaces for the same reason.

Where Cricket departs from a more traditional design is that we let clients directly access regular data via Mach's external pager facilities. (Bitmaps and other meta-data structures are still inaccessible, of course.) Although this compromises protection somewhat, we view it as manageable and worth the extra performance for the

types of applications we have in mind. Moreover, because all database accesses filter through Cricket's locking mechanism, which is discussed below, an application can only damage the data pages that it has gained access to anyway. Without direct access to the database, a client application would have to make an explicit request to read data into its address space, and it would have to take analogous steps to have it written back. This would involve added complexity, copying costs, extra buffering (possibly leading to double-paging [Bric76]), and would also destroy the abstraction of a single-level store.

5.1.3. Concurrency Control

By default, Cricket provides transparent, two-phase, page-level locking for client access to the database. This is done using Mach's exception handling facility [Blac88], which allows the exceptions of one task to be caught and handled by another task. In our case, Cricket handles exceptions for client tasks.

When a client first *connects* to Cricket, its exception handler is set to be the Cricket server. Later, when the client executes *begin_transaction*, all the virtual addresses in the client that map to the database are protected against read and write access. Subsequent attempts by the client to access a page in the database triggers an address exception, causing Mach to block the client and send a message to Cricket. The message is received by a Cricket thread, which attempts to acquire the appropriate (read or write) lock for the client, blocking itself if necessary. Once the lock has been acquired, the thread fixes the client's access permissions for the page via a kernel call and then lets Mach know that the exception has been successfully handled. At that point, Mach resumes the client.

The exchange of messages involved in catching an address exception and setting a lock is similar to that shown in Figure 1 for external pager fault handling. As one can imagine, setting a lock is not cheap! But compared to a more traditional design, our scheme is not as bad as it may appear at first glance. In a more traditional design, an RPC would typically have to be sent from the client to the database system to acquire a lock. And, as our preliminary results will show, exception handling in Mach is not drastically more expensive than sending an RPC.[2]

As mentioned earlier, we are also experimenting with different concurrency control options other than simple two-phase, page-level locking. Among other things, we eventually intend to support dirty reads [Moh89a] and also design- or file-level locks. The latter would be used by design transactions, where aborting a long-running transaction due to a deadlock makes little sense. Of course, the smallest granularity of locking that we can transparently provide in Cricket is limited to a page, but for the applications we have in mind that should be sufficient.

[2] It is worth noting that we also experimented with an alternative locking scheme where the exception handler ran as a thread in client's runtime support code. When an address exception was caught, this thread would send an RPC to Cricket to acquire the appropriate lock. Because of RPC costs, this turned out to be more expensive than the design we have chosen.

It is important to note that using address exceptions to trigger locking is not a new idea. Exceptions were also used in the Bubba database system [Bora90] to set locks. Our scheme differs from theirs in that we perform lock management in a user-level task, whereas locking was performed by the operating system in Bubba. This required special modifications to the operating system. Address exceptions have also been used by Li [Li86] to implement memory coherency in a distributed virtual memory system and in the language ML to trigger garbage collection [Appe86].

5.1.4. Disk Allocation

Cricket uses an extent-based scheme for managing disk space. A disk is partitioned into extents, with each extent containing the same number of pages — usually at least 16 Kbytes worth. Extents and the pages within an extent are allocated in a lazy manner, much like in Camelot. Linear hashing [Litw80] is used to map a virtual address to a physical extent on disk, allowing us to efficiently handle sparse databases. Because hashing is done on an extent basis, the hash table will generally consume very little space.

For allocating persistent data, we provide what amounts to an object-based version of Camelot's recoverable virtual memory. The runtime support code provides a *DBmalloc* function for allocating persistent "objects" and a corresponding *DBfree* for deallocating them. *DBmalloc* takes *size* and *near−hint* parameters. The *size* parameter tells how much space to allocate, while the *near−hint* parameter is a virtual memory address that tells *DBmalloc* where it should try to allocate space. The *near−hint* is used to simultaneously provide both virtual and physical clustering. That is, *DBmalloc* tries to allocate the new object on the same page as the *near−hint*. Failing that, it sends an RPC to Cricket, which tries to allocate the object either within the same extent as the *near−hint* or as physically close to it as possible. (Optimizations to cut down on RPCs are obviously possible here.)

As illustrated in Figure 3, individual disk pages are formatted as slotted pages [Date81]. The slot information at the bottom of a page is used to keep track of the objects and the free space on the page. When all the space on a page is free, it is marked as such in the page-allocation bitmaps maintained by the Cricket server. Large multi-page objects are allocated as runs of pages that are virtually contiguous, but not necessarily physically contiguous. Only the first page of a large object is formatted as a slotted page.

In addition to providing information about the objects on a disk page, the slot array at the bottom of a page also provides a level of indirection for accessing the objects on the page. If that extra level of indirection is always used, then it becomes possible to compact the free space on pages during idle periods. By default, this is not done because it would force applications to distinguish between non-persistent and persistent data access. However, in object-oriented languages that provide encapsulation, it may be possible to hide the extra level of indirection.

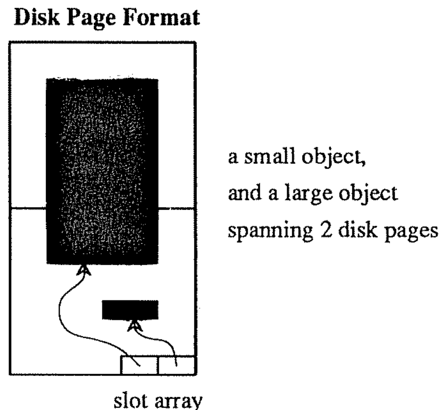

Figure 3: The Format of Disk Pages

5.1.5. Buffer Management

At the moment, we delegate all page replacement decisions for regular data to Mach. Consequently, an LRU replacement policy is used by default. For the types of applications we have in mind, where the working set of an application will typically fit in memory, this is expected to be adequate. As noted in [Eppi89], the beauty of letting Mach buffer regular data is that it effectively provides a buffer pool that dynamically changes its size in response to other system activity.

We examined two alternatives for managing system meta-data such as the page-allocation bitmaps. The first alternative was to maintain a small, wired-down virtual memory buffer pool in the Cricket server, while the second alternative was to map meta-data into the virtual address of Cricket itself and treat it as yet another memory object. We have chosen the first alternative because of the expense associated with using an external pager. Moreover, the abstraction of a single-level store is not particularly important for meta-data.

5.2. Unresolved Design Issues

5.2.1. Files

Although we recognize that files are needed to group related objects, we have not yet settled on a particular implementation for them. Given enough address bits (e.g., 64 bits), it may be sufficient in many cases to simply partition the virtual address space into large fixed-sized segments and treat each segment as a different file. Another alternative is to view a file as a list of (not necessarily contiguous) extents. This would require that all the objects in an extent belong to the same file.

5.2.2. Index Management

Eventually, we would like to include support for indexes such as B-trees in Cricket. An index in Cricket would simply map from some user-defined key to the virtual address of an object in the database. Our view is that indexes need to be managed by Cricket for reasons of protection and also performance.

As far as protection goes, we view indexes as meta-data, and as such they must be protected. An errant client application could cause considerable and potentially unrepairable damage if it were allowed write access to an index. And as far as performance goes, our feeling is that simple two-phase, page-level locking is inadequate for indexes, even in a design environment. Consequently, index pages cannot be treated as regular data. Obtaining adequate system performance usually requires fairly complex concurrency control and recovery algorithms to be used on indexes [Moh89b]. (In general, the same holds true for all meta-data structures.)

Index management presents something of a dilemma because on the one hand we would like to protect indexes from being damaged by client applications, but on the other hand the cost of sending an RPC to the Cricket server for each index access is likely to be too expensive, even if they are batched. To get around this dilemma, we are examining the possibility of giving clients read-only access to index pages. In this scheme, the runtime support code would take care of read operations on indexes (including locking), but updates would be forwarded (perhaps in batch-mode) to the Cricket server.

5.2.3. Recovery

Recovery is another area where we have yet to settle on a particular implementation. In discussing recovery algorithms, one of the key things to remember is that we give response time priority over throughput in Cricket. As a result, we are willing to accept a recovery algorithm that slows down transaction commit somewhat if it significantly improves response time during the execution of the transaction. Another thing to remember is that Cricket is intended to be used in a design environment, where the same set of persistent objects may be updated thousands of times by the same transaction. In such an environment, traditional old-value/new-value logging is clearly inappropriate.

At this point, we have decided that for disk allocation data, indexes, and all other meta-data, we will use the ARIES recovery algorithm [Moh89a], which is based on operation logging. For regular data, we have identified a number of alternatives, all of which require a *no-steal* buffer policy.[3] With a no-steal policy, steps must be taken to ensure that a dirty data page is not written to its home location on disk until the transaction that has modified the page commits. For the types of applications we have in mind, this is not expected to be a problem (especially in a

[3] Note that with a no-steal policy, logging is only needed to provide commit atomicity and to support recovery from media failure.

distributed environment, which is briefly discussed below). The advantages of using a no-steal policy are that old-values do not have to be logged and repeated changes are aggregated before being logged at commit.

One alternative for regular data recovery in Cricket is to simply log full pages at commit. Although this sounds like it could generate excessive amounts of log data, the applications that use Cricket may tend to update a large fraction of each page that they modify. If this turns out to be the case, then logging full pages at commit will result in an efficient recovery algorithm. During idle periods, the on-line log can be compressed by removing all but the most recent copy of a given page.

Another alternative is to use a copy-on-write mechanism. When a write lock for a page is granted, the page is copied to a temporary location in memory. Then, at commit, the new version of the page is compared to its original and the changed portions are logged. If log space is a concern, a compression algorithm can be applied to the log records that are generated.

One final alternative is to require all updates to persistent objects to filter through a runtime support function. The support function would record information that indicates which persistent objects have been modified. At commit, the Cricket server would then use the recorded information and its knowledge of which pages were modified to generate new-value log records. The disadvantage of this approach is that persistence is no longer transparent to applications.

5.2.4. Moving to a Distributed Environment

Since a client/server hardware configuration is expected to be the norm for design applications, we naturally plan on moving Cricket to a distributed environment. In fact, that has always been our main goal, and the single-site architecture is really just a stepping stone. Because it has been built on top of Mach, client applications and the Cricket server can already run on separate machines. However, the current design has not yet been optimized for the distributed case.

When we move to a distributed environment, we expect Cricket's architecture to look like Figure 4. As illustrated, Cricket will be split into a front-end and a back-end. The front-end will take care of functions that can be handled more efficiently on the local machine, while the back-end will take care of global functions like cache coherency. Note that because we provide a single-level store to clients, this architecture supports what amounts to distributed, transactional, shared, persistent virtual memory (phew!). Although we could use the algorithms described by Li [Li86] to maintain memory coherency across machines, transaction semantics open up the possibility for us to use more efficient algorithms. There has been some work done in this area (see [Wilk90, Dewi90]), but not in the context of a single-level store. One of Cricket's designers is actively working on this problem already [Fran90].

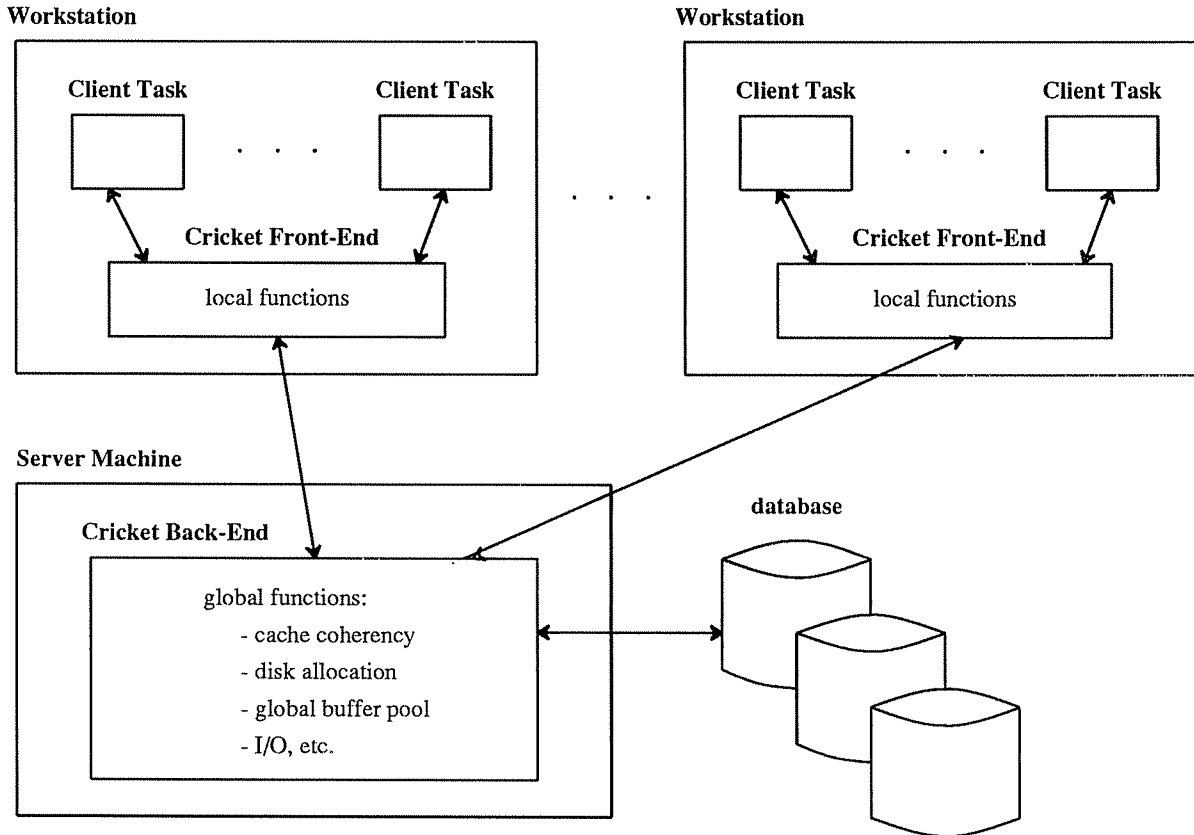

Figure 4: Distributed Cricket

Some of the interesting problems that surface in a distributed environment include index management, buffering, and the general question of what functionality belongs in the front-end and what functionality belongs in the back-end. We also expect distribution to affect our choice of recovery algorithms. For example, in a distributed environment, it probably makes more sense to offload as much commit processing as possible to the front-end machine. Also note that there is no need to use Mach's external pager facilities in the back-end, as the data pages that are cached there are not directly accessible to clients.

6. PRELIMINARY PERFORMANCE RESULTS

To get a rough idea of how the single-site version of Cricket can be expected to perform, we ran a series of benchmarks on a DEC MicroVax 3200 workstation with 16 Mbytes of memory. The benchmarks were run in single-user mode on version 2.5 of Mach with the workstation disconnected from the network. We only ran single-user benchmarks, and the average cost of a given operation was calculated by performing the operation several thousand times, and then dividing the measured elapsed time by the number of operations performed. This was done several times to check for stability, and the average observed values are the ones reported here. The Mach real-time clock, which has a resolution of roughly 17 msec, was used to measure elapsed times. The virtual page

size in the version of Mach that we were running under was 4 Kbytes.

Before running any benchmarks, we measured the CPU costs of various key operations in Mach. This was done to get a general feel for the cost of different operations on the MicroVax. The results are presented in Table 2.

Operation	Cost in usec
getpid( )	108
vm_protect( )	490
bcopy( ) a 4 Kbyte page	585
null RPC	1,275
send page out-of-line	1,316
send page in-line	4,493

Table 2: Cost of Various Operations in Mach

Getpid (get process ID) is the simplest kernel call that we could think of, while *vm_protect* is the call that Cricket uses to set the access rights on client pages. *Bcopy* is a library function for copying data in memory, and the last three operation costs listed are for sending a null RPC with no arguments, sending a page out-of-line via an RPC, and sending a page in-line via an RPC, respectively.

6.1. The Cost of Using Mach's External Pager and Exception Handling Facilities

The performance of Cricket is largely dependent on the cost of Mach's external pager and exception handling facilities. To measure these costs, we used a simple scan benchmark. In this benchmark, a single client *connects* with Cricket, invokes *begin_transaction* and then sequentially touches the first 1,280 pages (i.e., 5 Mbytes) of the mapped database. All I/O is short-circuited in the benchmark by having the Cricket server pass Mach a pointer to a dummy page in *memory_object_data_provide*. By using the scan benchmark and by turning off all aspects of transaction management in Cricket other than exception handling and external pager requests, we were able to obtain the results shown in Table 3. These results capture the per-page CPU cost of using Mach's external pager and exception handling facilities.

The costs listed in Table 3 are as follows: (1) is simply the CPU cost of handling a page fault for a page that is already cached in Mach kernel memory. (2) is the CPU cost of handling an address exception in Cricket to trigger locking on a database page. (3) is the CPU cost of having the kernel send a *memory_object_data_request* message to Cricket, with Cricket responding via a *memory_object_data_provided* message. (4), which should approximately equal (1) + (2), is the CPU cost that a client incurs on the first access to a database page that is cached in kernel memory. Finally, (5), which should approximately equal (1) + (2) + (3), is the CPU cost that a client incurs on the first access to a database page that is not cached in kernel memory.

Event	Cost in usec
(1) page fault that is handled completely in the kernel	420
(2) handle address exception in Cricket	3,180
(3) *memory_object_data_request* & *memory_object_data_provided*	3,221
(4) page fault + address exception	3,605
(5) page fault + address exception + *memory_object_data_request* & *memory_object_data_provided*	6,845

Table 3: Per-Page CPU Cost of External Pager and Exception Handling Facilities

As Table 3 clearly indicates, the external pager and exception handling facilities of Mach are not exactly free! Most of the expense presumably comes from context switches, message costs, and management of the kernel data structures associated with memory objects. However, the reader should bear in mind that (4) or (5) will only be incurred on the first access to a page. Furthermore, there are a number of ways that these costs can be reduced. One way is to simply do large block-sized I/O operations for regular data. We simulated the effect that this would have on the CPU costs in Table 3 by providing 16 Kbyte blocks of data to Mach in *memory_object_data_provided.* When this was done, the CPU cost of (5) dropped to 4,954 usec per 4 Kbyte page. Another method to reduce costs is to read data from disk and asynchronously call *memory_object_data_provided* as soon an address exception for an uncached data page is caught.[4] This is in contrast to waiting for an explicit *memory_object_data_request* message from the Mach kernel. Finally, large-grained locks can be used to cut down on the number of exceptions generated. To examine the effects of combining these methods, we simulated doing 16 Kbyte block I/O as soon as an address exception was generated for the first page in the block, and we also set the granularity of locking to 4-page units. When this was done the CPU cost of (5) dropped further to 2,248 usec per 4 Kbyte page. In design environments, we may do even better if design-level locks are acceptable.

6.2. Comparing Cricket to a General-Purpose Database Storage System

To determine how Cricket's performance compares to a general-purpose database storage system, we ran a tree-search benchmark on Cricket and also on the single-user version of the EXODUS Storage Manager [Care86]. In this benchmark, a persistent tree is searched in a depth-first manner, and the number of times the tree is searched can be varied. No processing is done on a node other than to follow its edges to neighboring nodes. This benchmark was chosen because it is representative of the types of data access that Cricket applications are expected to make. It is important for readers to realize that this is not really a fair comparison, as the single-user EXODUS Storage Manager does not provide shared access, protection, or locking. Nonetheless, we were still able to obtain some results that we thought might be of interest to other researchers.

[4] This is a rather obvious thing to do, but surprisingly Mach does not currently provide a way for an external pager to determine if a given page of a memory object is cached in kernel memory. Hopefully, this design flaw will be fixed in the near future.

In the tree-search benchmark, we used a tree with a depth of 4 and a node fanout of 11 (16,105 nodes total). For uniformity, the nodes in the tree were padded so that data pages in both the EXODUS Storage Manager and Cricket contained the same number of nodes, namely 12. As a result of padding nodes, the tree spanned 1,343 pages in both storage systems. Readers should note that padding the nodes in this manner biases the results in favor of the EXODUS Storage Manager due to the fact that its object identifiers consume 12 bytes, whereas they only consume 4 bytes in Cricket. Therefore, under normal circumstances, the resulting tree would tend to span fewer disk pages in Cricket than than it does in the EXODUS Storage Manager. This would in turn lead to less I/O, smaller buffering requirements, etc.

To measure the effect of doing I/O, we used a version of the tree-search benchmark that read the tree from disk at startup. When we sat down and looked at the numbers that were generated for Cricket, however, they made no sense. In particular, the CPU cost of using Mach's external pager facilities was not showing up. A little experimentation revealed that Mach could not issue I/O requests fast enough to avoid rotational delays (even for sequential reads on a raw disk partition). Since the average rotational delay for the disk we used was 8.3 msec, this meant that Cricket could, on average, do an extra 8.3 msec of CPU processing per page without it ever showing up in a single-user benchmark! This, of course, lead to the strange results. In the near future, we will try to get I/O numbers using some other benchmark.

As a result of the above problems and due to time constraints, we ended up generating only the results shown in Table 4. To ensure that no I/O would take place, the whole tree was read into memory by a separate transaction before the benchmark was run. Although exception handling for locks was turned on in Cricket, the transaction management code associated with locking was turned off to keep the comparison as fair as possible.

Setting	Elapsed Search Time in msec One Pass	Elapsed Search Time in msec Two Passes
non-persistent tree	789	1599
Cricket	5,680	6,515
EXODUS	6,230	12,467

Table 4: Results for the Tree-Search Benchmark

The first line in Table 4 shows the cost of searching a non-persistent version of the tree. The values shown there and in Table 3 can be used to validate the results that we obtained for Cricket. For example, the elapsed time for the one-pass search in Cricket is estimated to be 5,631 msec (789 msec for the base cost of executing the search code, plus 4,842 msec for the cost of handling page faults and exceptions on 1,343 pages). This estimate is quite close to the measured time of 5,680 msec. The same holds true for all the results, and therefore we are confident that the numbers we obtained are accurate.

As the results in Table 4 show, even with just 12 nodes per page and with almost no processing on a node, Cricket was still able to outperform the EXODUS Storage Manager. To understand why, it suffices to look at the interface that the EXODUS Storage Manager provides for accessing persistent objects. There, access to a persistent object is obtained via the *ReadObject* procedure call, which locates the object in the buffer pool, pins the page that contains it, and sets up an indirect pointer that is used to access the object. After the object is no longer needed a *ReleaseObject* call is issued to upin the object. In the tree-search benchmark, *ReadObject* and *ReleaseObject* had to be called once per node per search pass. It was primarily the costs associated with these two procedures (somewhere around 330 usec for the pair) that lead to the slower times in the EXODUS Storage Manager. Given that this benchmark was somewhat biased in favor of the EXODUS Storage Manager, we view these results very positively. To us, they suggest that for its intended applications, Cricket can provide better performance than a general-purpose database storage system.

7. RELATED WORK

The work most closely related to ours is that done by the implementors of the Bubba database system at MCC [Bora90, Cope90]. In Bubba, the kernel of an AT&T UNIX System V kernel was modified to provide a single-level store with automatic, two-phase, page-level locking. Although we have borrowed a number of ideas from Bubba, several differences distinguish Cricket from the approach taken in Bubba. One difference is that Bubba's implementors had to modify the operating system kernel, since they did not have the luxury of using Mach. This, of course, caused problems with portability. Also, their recovery algorithms relied on battery-backed RAM, again causing problems with portability. Furthermore, in contrast to Cricket, the implementors of Bubba were able to ignore protection issues because their applications were written in FAD, which is a "safe" language. Finally, the focus in Bubba was on building a highly parallel database system, whereas in Cricket we are more interested in storage system issues, client/server hardware configurations, and providing support for design environments and persistent languages.

The Camelot Distributed Transaction System [Spec88] is another related work. Camelot also used the external pager facilities of Mach to provide a single-level store. In contrast to Cricket, however, the single-level store that Camelot provides is not meant to be directly accessed by client applications. Instead, it is intended to be accessed only within a "data server" for storing all the persistent data and meta-data managed by that server. It is not clear, however, that the abstraction of a single-level store is all that important in the context of a data server. In contrast to Cricket, Camelot also provides fairly conventional locking and recovery services that must be explicitly invoked by its clients.

The last related work that we need to mention is that done in IBM's 801 prototype hardware architecture [Chan88]. In the 801 prototype, the operating system essentially provided mapped files with automatic concurrency control and recovery. Special hardware was added for both locking and logging. While this is an interesting approach, our view is that it suffers from being too inflexible. In particular, no support was given for anything other than two-phase locking and value-based logging. This, of course, causes problems for indexes and other meta-data structures where two-phase locking is inappropriate. Distribution is also problem. Finally, special hardware support was necessary, which clearly causes problems with portability.

8. CONCLUSIONS

In this paper, we have introduced Cricket, a database storage system that is intended to be used as a platform for design environments and persistent programming languages. Cricket uses the memory management primitives of the Mach operating system to provide the abstraction of a shared, transactional single-level store that can be directly accessed by user applications. In the paper, we described our motivation for building Cricket, and we argued that a single-level store is a useful abstraction for many database applications. We also presented a fairly detailed description of Cricket's architecture, outlining a single-site architecture as well as a distributed architecture that we will eventually move to. Finally, we presented some preliminary performance results, which compared Cricket to the EXODUS Storage Manager. A simple tree-search benchmark was used to show that, for its intended applications, Cricket can provide better performance than a general-purpose database storage system.

As far as the implementation status of Cricket is concerned, the single-site version prototype is currently up and limping along. However, much work remains. We have stolen the code for transaction management and locking from the EXODUS Storage Manager, but recovery has yet to be implemented; and likewise for index management. Eventually, of course, we will move to a distributed architecture. That move looks like it will lead to a number of interesting research problems. These include the problem of how to split storage system functionality between client machines and servers, and also the problem of maintaining memory coherency for what amounts to distributed, transactional, shared, persistent virtual memory.

REFERENCES

[Acce86] M. Accetta et al., "Mach: A New Kernel Foundation for UNIX Development," *Proc. of the Summer Usenix Conf.*, June 1986.

[Agra89] R. Agrawal and N. Gehani, "ODE (Object Database and Environment): The Language and the Data Model," *Proc. of the 1989 ACM SIGMOD Conf.*, June 1989.

[Appe86] A. Appel et al., "Garbage Collection Can be Faster Than Stack Allocation," Computer Science Tech. Report 045-86, Princeton Univ., June 1986.

[Atki87] M. Atkinson and P. Buneman, "Types and Persistence in Database Programming Languages," *ACM Computing Surveys*, 19(2), 1987.

[Bako90] H. Bakoglu et al., "The IBM RISC System/6000 Processor: Hardware Overview," *IBM Journal of Research and Development*, 34(1), 1990.

[Bens72] A. Bensoussan et al., "The Multics Virtual Memory: Concepts and Design," *CACM*,

[Bora90] H. Boral et al., "Prototyping Bubba, A Highly Parallel Database System" *IEEE Trans. on Data and Knowledge Eng.*, 2(1), 1990. 15(5), May 1972.

[Blac88] D. Black et al., "The Mach Exception Handling Facility," Computer Science Tech. Report 88-129, Carnegie Mellon Univ., April 1988.

[Bric76] P. Brice and S. Sherman, "An Extension of the Performance of a Database Manager in a Virtual Memory System using Partially Locked Virtual Buffers," *ACM Trans. on Database Systems*, 6(1), 1976.

[Care86] M. Carey et al., "Object and File Management in the EXODUS Extensible Database System," *Proc. of the 12th Intl. Conf. on Very Large Databases*, Sept. 1986.

[Chan88] A. Chang and M. Mergen, "801 Storage: Architecture and Programming," *ACM Trans. on Computer Systems*, 6(1), 1988.

[Chan89] E. Chang and R. Katz, "Exploiting Inheritance and Structure Semantics for Effective Clustering and Buffering in an Object-Oriented DBMS", *Proc. of the 1989 ACM SIGMOD Conf.*, June 1989.

[Chou85] H-T. Chou and D. Dewitt, "An Evaluation of Buffer Management Strategies for Relational Database Systems," *Proc. of the 1985 VLDB Conf.*, Aug. 1985.

[Cock84] W. Cockshott et al., "Persistent Object Management Systems," *Software-Practice and Experience*, vol. 14, 1984.

[Cope90] G. Copeland et al., "Uniform Object Management," *Proc. of the Intl. Conf. on Extending Database Technology*, March 1990.

[Date86] C. Date, "An Introduction to Database Systems," Ch. 3., pg. 56, Addison-Wesley, Reading Mass. 1986.

[DBE87] *Database Engineering*, Special Issue on Extensible Database Systems, M. Carey ed., 10(2), June 1987.

[DeWi90] D. DeWitt et al., "A Study of Three Alternative Workstation-Server Architectures for Object-Oriented Database Systems," Computer Science Tech. Report 907, Jan. 1990. Univ. of Wisconsin,

[Drav88] R. Draves and E. Cooper, "C Threads," Computer Science Tech. Report 88-154, Carnegie Mellon Univ., June 1988.

[Duch89] D. Duchamp, "Analysis of Transaction Management Performance," *Proc. of the 11th Symposium on Operating System Principles*, Dec. 1989.

[Eppi89] J. Eppinger, "Virtual Memory Management for Transaction Processing Systems," Ph.D thesis, Computer Science Tech. Report 89-115, Carnegie Mellon Univ., Feb. 1989.

[Ford88] S. Ford et al., "ZEITGEIST: Database Support for Object-Oriented Programming," *The 2nd Workshop on Object-Oriented Database Systems*, 1988.

[Fran90] M. Franklin, et al. Paper in progress on algorithms for maintaining cache coherency in a client/server hardware environment. Univ. of Wisconsin.

[Horn87] M. Hornick and S. Zdonik, "A Shared, Segmented Memory System for an Object-Oriented Database," *ACM Trans. on Office Information Systems*, 5(1), 1987.

[Katz87] R. Katz and E. Chang, "Managing Change in a Computer-Aided Design Database," *Proc. of the 1987 VLDB Conf.*, Sept., 1987

[Kern78] B. Kernighan and D. Ritchie, "The C Programming Language," Prentice-Hall, 1978.

[Li86] K. Li and P. Hudak, "Memory Coherence in Shared Virtual Memory Systems," *Proc. of the 5th Annual ACM Symposium on Principles of Distributed Computing*, Aug. 1986.

[Litw80] W. Litwin, "Linear Hashing: A New Tool for File and Table Addressing," *Proc. of the 1980 VLDB Conf.*, Aug. 1980.

[Lind87] B. Lindsay et al., "A Data Management Extension Architecture," *Proc. of the 1987 ACM SIGMOD Conf.*, May 1987.

[Maho86] M. Mahon et al., "Hewlett-Packard Precision Architecture: The Processor," *Hewlett-Packard Journal*, August 1986, pp. 4-22.

[Maie89] D. Maier, "Making Database Systems Fast Enough for CAD Applications," in *Object-Oriented Concepts, Database and Applications*, W. Kim and F. Lochovsky, eds., Addison-Wesley, 1987, pp. 573-581.

[Moh89a] C. Mohan et al., "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging," IBM Research Report RJ6649, Jan. 1989.

[Moh89b] C. Mohan and F. Levine, "ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging," IBM Research Report RJ6846, Aug. 1989.

[Moss88] J. Moss and S. Sinofsky, "Managing Persistent Data with Mneme: Designing a Reliable, Shared Object Interface," in *Advances in Object-Oriented Database Systems*, vol. 334 of *Lecture Notes in Computer Science*, Springer-Verlag, 1988, pp. 298-316.

[Moss90] J. Moss, "Working with Persistent Objects: To Swizzle or Not to Swizzle," Computer Science Tech. Report 90-38, Univ. of Massachusetts, May 1990.

[Rich89] J. Richardson, *E: A Persistent Systems Implementation Language*, Ph.D thesis, Computer Science Tech. Report 868, Univ. of Wisconsin, August 1989.

[Sche90] H. Schek et al., "The DASDBS Project: Objectives, Experiences, and Future Perspectives," *IEEE Trans. on Data and Knowledge Eng.*, 2(1), 1990.

[Schu90] D. Schuh et al., "Persistence in E Revisited — Implementation Experiences," *Proc. of the 4th Intl. Workshop on Persistent Object Systems Design, Implementation and Use*, Sept. 1990.

[Spec88] "The Guide to the Camelot Distributed Transaction Facility: Release 1," A. Spector and K. Swedlow eds., Carnegie Mellon Univ., 1988.

[Ston81] M. Stonebraker, "Operating System Support for Database Management," *CACM*, 24(7), 1981.

[Ston84] M. Stonebraker, "Virtual Memory Transaction Management," *ACM Operating Systems Review*, 18(2), 1984.

[Ston90] M. Stonebraker et al., "The Implementation of POSTGRES," *IEEE Trans. on Data and Knowledge Eng.*, 2(1), 1990.

[Trai82] I. Traiger, "Virtual Memory Management for Database Systems," *ACM Operating Systems Review*, 16(4), 1982.

[Wilk90] K. Wilkinson and M. Neimat, "Maintaining Consistency of Client-Cached Data," *Proc. of the 1990 VLDB Conf.*, Aug. 1990.

[Youn87] M. Young et al., "The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System," *Proc. of the 11th Symposium on Operating System Principles*, Nov. 1987.