801 Storage: Architecture and Programming

ALBERT CHANG and MARK F. MERGEN
IBM T. J. Watson Research Center

Based on novel architecture, the 801 minicomputer project has developed a low-level storage manager that can significantly simplify storage programming in subsystems and applications. The storage manager embodies three ideas: (1) *large virtual storage*, to contain all temporary data and permanent files for the active programs; (2) the innovation of *database storage*, which has implicit properties of access serializability and atomic update, similar to those of database transaction systems; and (3) access to all storage, including files, by the usual operations and types of a high-level *programming language*.

The IBM RT PC implements the hardware architecture necessary for these storage facilities in its storage controller (MMU). The storage manager and language elements required, as well as subsystems and applications that use them, have been implemented and studied in a prototype operating system called CPR, that runs on the RT PC. Low cost and good performance are achieved in both hardware and software. The design is intended to be extensible across a wide performance/cost spectrum.

Categories and Subject Descriptors: B.3.2 [**Memory Structures**]: Design Styles—*virtual memory*; C.1.1 [**Processor Architectures**]: Single Data Stream Architectures—*reduced instruction set computers*; D.3.3 [**Programming Languages**]: Language Constructs; D.4.2 [**Operating Systems**]: Storage Management; D.4.3 [**Operating Systems**]: File Systems Management; D.4.7 [**Operating Systems**]: Organization and Design

General Terms: Algorithms, Design, Experimentation, Performance

Additional Key Words and Phrases: Atomic commit, locking, protection, relocation, segmentation, transaction

1. INTRODUCTION

This paper presents storage architecture and associated programming of the 801 minicomputer project. The 801 [17] is a prototype reduced instruction set computer (RISC) [16], though our storage work is mainly independent of this. 801 storage architecture contains three necessary features: *segment registers* that expand virtual addressing beyond 32 bits, an *inverted page table* for address translation, and a *transaction locking mechanism* that causes hardware calls to a software lock manager. The IBM RT PC [12, 19] implements these features in its hardware storage controller (MMU) [11].

CPR (Control Program Research) is a prototype operating system used for research in several areas, with major emphasis on storage facilities. After earlier

Authors' address: IBM T. J. Watson Research Center, Yorktown Heights, NY 10598.

development on 801 simulators and prototype hardware, it now runs on the RT PC. We describe only its virtual storage and transaction features, the storage manager that implements them, and related issues.

The PL.8 compiler [3] is also part of the 801 project. PL.8 is a PL/1 dialect used mainly for systems programming. The compiler also accepts Pascal and C programs, and produces code for several machines, including prototype 801 and the RT PC. We describe PL.8 language and compiler function added to improve programming with files.

This paper is organized as follows: Section 2 reviews background and previous work. Section 3 covers the storage architecture, including 801 addressing and the relocation mechanism. Section 4 describes how CPR uses the storage architecture, and Section 5 describes the programming language interface to the storage system. Section 6 provides implementation details, and Section 7 contains performance data. Section 8 contains a summary.

2. BACKGROUND

There are two major themes that have influenced the development of 801 storage: *virtual storage*, particularly virtual addressing of permanent files, and the *transaction* concept. The idea that permanent files should be placed in virtual storage, along with temporary data and programs, was first suggested and applied by Multics [4, 6], with claimed benefits of easier programming and better sharing. This approach is sometimes called a *one-level store* because all data in the system may be directly addressed by the storage instructions of the processor. Programming is simplified because the operating system, driven primarily by missing page interrupts, performs all the I/O operations and buffer management needed to move data, including files, between levels of the storage hierarchy. Sharing is better because processes access one copy of files in virtual storage, rather than separate copies they each read from disk. Real storage becomes a systemwide shared buffer pool. Contention between buffering and virtual paging is eliminated. Adding real storage improves this buffering. More recently, the midrange IBM System/38 [10] and the Pilot system [13, 18] for a Xerox personal computer have applied similar approaches for similar reasons.

The transaction concept was originally developed in production and research database management systems (e.g., IMS [14] and System R [1]), and has since been widely studied. Informally, a transaction is a sequence of actions grouped together that observe some composite state of the database and/or that change it from one state to another. Usually database managers allow different transactions to share the database and run concurrently, with protection provided (commonly by use of locking and change logging) against undesired interactions and failures. The strongest properties are those of *serializable* transactions with *atomic update* and *permanence* [8, 9]. These transactions appear to run one after another in some order rather than interleaved and, in case of failures, either all or none of the actions of each transaction appear permanently in the database. Serializable atomic transactions are a powerful tool to maintain database consistency. If the actions of each transaction produce a consistent state and the database manager guarantees the above execution properties, then the database will always be consistent as each transaction begins. After recovery from most failures (system, deadlock, etc.), transactions that did not complete because of the failure may simply be restarted from their beginnings.

For some time there has been discussion about possible relationships between virtual storage, database management, and transaction facilities in an operating system [22–25]. Operating systems that include low-level transaction facilities have appeared (e.g., Camelot [21] and LOCUS [26]). They suggest that transactions have applicability apart from traditional database management, as in distributed systems and in management of operating-system data such as directories.

With 801 we add the idea of hardware that *automatically* invokes the transaction mechanisms of locking and change logging, at relatively fine granularity, to simplify programming and enforce correctness. This is similar to the use of hardware page faults to implement virtual storage. Stonebraker [23] has also suggested a hardware approach to page locking.

We simply accept that this background describes an attractive set of ideas and potential benefits. To us the important question is: can a system that combines virtual storage access to files with implicit transaction functions be implemented with competitive cost/performance? Our research indicates a promising approach.

3. STORAGE ARCHITECTURE

3.1 Segment Registers

The 801 and many other contemporary micro- and minicomputers have 32-bit general-purpose or base registers and compute 32-bit storage addresses. We want to have all the programs, temporary data, and files of a process directly addressable in virtual space. Thirty-two-bit addresses may not be sufficient for even one process if it is accessing large databases. Therefore, the hardware provides 16 segment registers to expand the virtual address space.

As shown in Figure 1, the upper 4 bits of a 32-bit *short address* name a segment register. The register provides a *segment id* that replaces the 4 bits of short address to form a *long virtual address*. This effectively creates a single long-virtual-address space, as shown in Figure 2, with segment boundaries at multiples of 256 Mbytes. The RT PC has a 12-bit segment id (4096 segments) and a 40-bit long virtual address. A larger virtual space could be achieved with more segment id bits (wider segment registers).

Each segment register also contains two bits called S and K that control storage protection for the segment. When the S bit is 1, the transaction locking mechanism is in effect for the segment, otherwise more conventional page protection applies. K is used only with page protection. More details are given later.

A single segment size of 256 Mbytes was chosen for three reasons: First, even quite large files will fit in a single segment. Very large objects, for example, databases, may require multiple segments, but are usually managed by subsystems that expect to use multiple files anyway for composite objects, for example, data and indices. Second, having one maximum size simplifies segment allocation, even though it wastes addressing bits if most segments are much smaller. Third, only 16 registers are required in hardware logic. Also, this is a modest addition

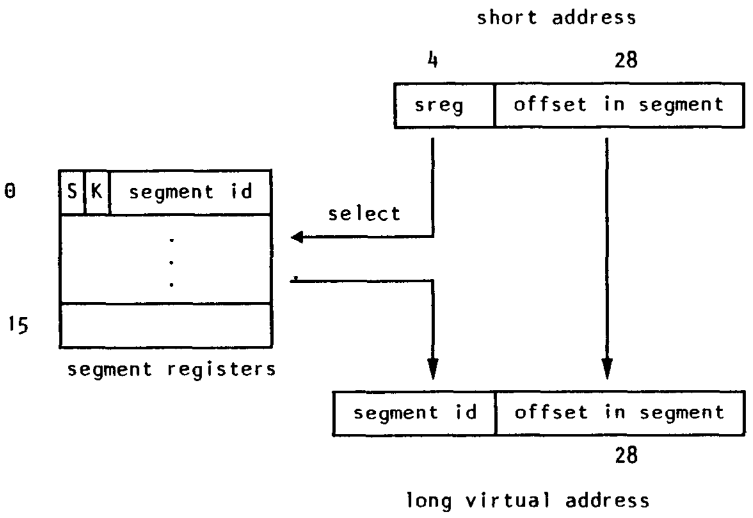

Fig. 1. Generation of a long virtual address.

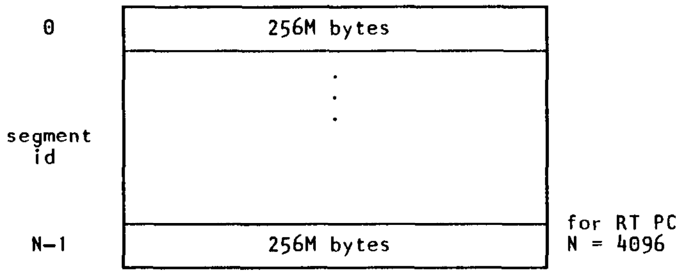

Fig. 2. The single long-virtual-address space.

to the state to be changed at process switch, when compared with 32 general registers of the prototype 801 plus floating-point registers and other state required in actual systems.

Segments are a convenient unit of access control and sharing between processes, as fully discussed in [4]. In 801, *load segment register* is a privileged instruction, so the supervisor may limit access by segment id to processes with proper rights. If several processes are allowed to load the same segment id, they directly share data in the single large virtual space, even if they use different segment registers. This means that 801 permits aliasing by short addresses, but not by long addresses that uniquely name virtual storage and are used in address translation.

An alternative to segments is a separate virtual space for each process and a page table for each space. The supervisor allows translation to real address only for pages containing data that a process has rights to access, but there are no segment boundaries in the virtual space. Sharing occurs when several processes have translations of the same or different virtual addresses to the same real addresses. We think this approach has many drawbacks. As previously stated, a 32-bit address space may not be sufficient for even one process. Allocation of variable sized areas to use this space efficiently is complicated, and such areas must be moved (change address) to expand. In general, it is not possible to coordinate virtual allocation between different processes. This complicates bookkeeping for sharing and causes aliasing, which has two important consequences in hardware design [20]: Multiple entries for the same data in a translation look-aside buffer may reduce the hit ratio, and virtual addressing in a storage cache directory generally cannot be used.

The expansion of addressing by segment registers is similar to that found in the Intel 80286 and predecessors, which expand a 16-bit address to 20 or 24 bits. However, they name the segment register by implication for some addresses or by specific bits in the machine instruction for others, rather than by bits of the computed address as in 801. The 801 approach has two advantages: First, a short parameter address passed to a subprogram implies which segment register to use. Second, two or more segments may be combined into one larger space, for purposes of address computation, by loading their segment ids into adjacent segment registers.

The single large-virtual-address space (single space of segment ids) is similar to that of System/38, although the latter has two segment sizes (16 Mbytes and 64 kbytes) and a longer virtual address (48 bits) than the RT PC. We chose a single larger segment size for reasons already given. Also, 801 long address generation is in the simple hardware spirit of RISC, while the System/38 generates long addresses from machine instructions in microcode. By contrast, the hardware of Multics has multiple spaces of segment numbers (the software uses one space for each process). A shared segment can have different segment numbers in different spaces. This forces undesirable aliasing, as mentioned previously.

3.2 Inverted Page Table

Pages are a convenient unit of real storage allocation and disk I/O [4]. Only currently referenced pages, rather than whole segments, need be in real storage. The single virtual space with large segments, described above, however, would be less attractive if large page tables were needed for address translation. As in System/38, an inverted page table avoids this difficulty. The idea of an inverted page table is that each entry represents one page of real storage and contains the address of the virtual page currently allocated to that real page. Given a virtual address to be translated, hardware searches the table for that address and, if found, uses the table index of the matching entry as the address of the desired real page.

Because hash searching is used, the inverted page table is really two tables, as shown in Figure 3, which must be in real storage. The page table has an entry for each real page. The hash table has a number of entries that is a power of two, equal to or greater than the number of page table entries. Translation uses the hash table and only the *segment id*, *vpage*, and *chain* fields of the page table. Remaining fields control locking and protection. A hash value is computed for each page table entry: the exclusive-or of the *segment id* and *vpage* fields. A hash

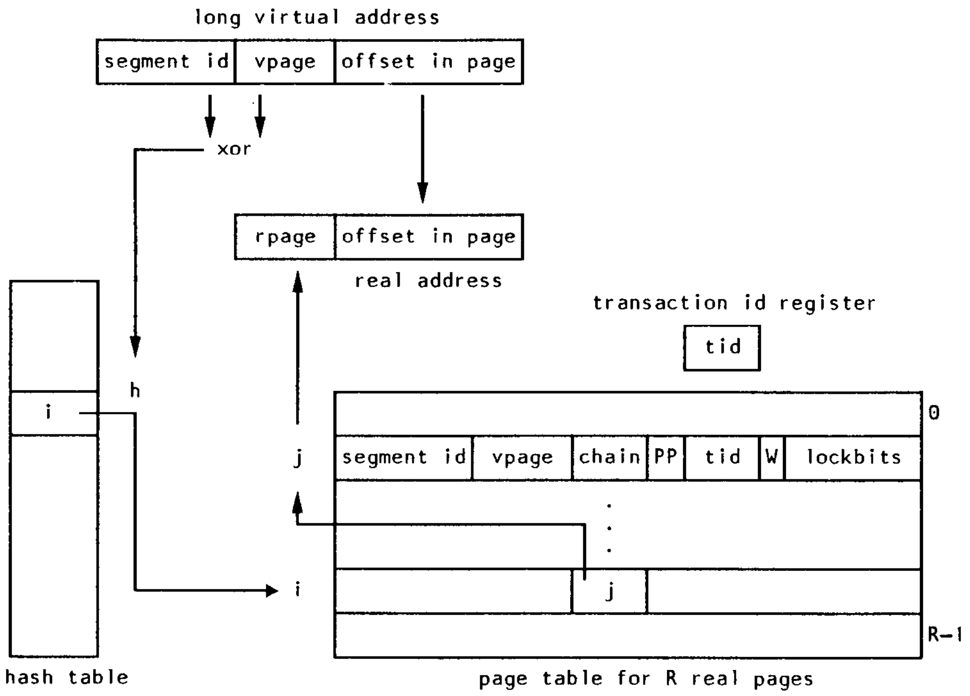

Fig. 3. The inverted page table.

value selects a hash table entry that points to the first (maybe only) page table entry with that hash value. Page table entries with equal hash values are chained together.

Figure 3 also shows an example of translation. Hash value *h*, computed from the long virtual address, is the index of a hash table entry. The value in the hash entry is *i*, the index of a page table entry. The *segment id, vpage* fields in this entry do not match those of the long virtual address, so the *chain* value *j*, the index of another page table entry, is used. This entry matches the long virtual address, so the value *j* becomes the *rpage* field in the real address. This translation requires three storage accesses.

To avoid storage access for each translation, the hardware first searches a conventional translation look-aside buffer (TLB) [20]. If not found there, translation by the inverted page table is attempted and, if successful, the translation is loaded into the TLB. If a null-value hash entry or *chain* field is encountered before a match is found, the hardware causes a page fault interrupt.

Traditional page tables have size related to total virtual space, with an entry per virtual page containing the real page address. Such tables are often two-level trees, with a page table per segment, requiring only recently referenced tables in real storage and 2 storage accesses per translation. An inverted page table has size related to real space, independent of total virtual space, and table space management is unnecessary. Assuming a uniform random distribution of virtual addresses, the RT PC expects 2.5 storage accesses per translation. If hash table size were doubled, this would decrease to 2.25 storage accesses.

3.3 Transaction Locking and Protection

When segment register S bit is 1 (Figure 1), the transaction locking mechanism applies to the segment, controlled by a *transaction id* register and the *tid*, *W*, and *lockbits* fields in each page table entry (Figure 3). The transaction id register must match the *tid* field to allow access in the page. The W bit determines if lockbits represent write locks or read locks. There is a lockbit for each *line* of 128 bytes in the page. The 2 kbyte pages of the RT PC require 16 lockbits per page. Read access to (load from) line k is permitted if lockbit(k) is 1 or if W is 1. Write access to (store into) line k is permitted only if lockbit(k) and W are both 1. If access is not permitted, by transaction id mismatch or by the lockbits, the hardware causes a lock fault interrupt. The read/only or read/write access permitted to line k may be summarized:

tid compare	lockbit(k)	$W = 0$	1
equal	0	—	r/o
equal	1	r/o	r/w
unequal	0/1	—	—

Each page table entry contains the locks of one transaction in that page, identified by the *tid* field. Locks of other transactions in that page must be recorded elsewhere. Only the transaction id register must be changed when the currently running transaction changes. The required match with the *tid* field prevents the current transaction from accessing pages whose table entries contain the locks of others. When transaction mismatch interrupt occurs, software must switch the locks in the page table entry to those of the current transaction.

Since there is only one lockbit per line, it is not possible to represent simultaneously the read and write locks of a transaction in the same page. Therefore, when lockbits are write locks (W is 1), the transaction may read any line, which is an implicit read lock on the whole page. Usual lock serialization prohibits granting any write locks in that page to other transactions. This means that a page may be shared by one writer and any number of readers whose read locks are disjoint from the write locks of the writer. Multiple writers in a page could be allowed if either lockbits per page or line size were doubled. We chose one writer to get 128-byte granularity with fewer lockbits.

Stonebraker [23] has proposed somewhat different locking assist hardware. It provides page locking with 4 bits and a count field per page. An array of 2 of these bits for all pages must be swapped during process switch, and page differences must be computed to achieve a compact log. Our approach provides finer granularity locks, which also allow logging less than full pages, and only the transaction id register must be changed during a process switch.

For segments that do not require transaction locking, for example, temporary storage, conventional page protection may be selected by setting segment register S bit (Figure 1) to 0. Segment register K bit and two PP bits in each page table entry (Figure 3) permit read/only or read/write access to pages as follows:

	$PP = 00$	01	10	11
$K = 0$	r/w	r/w	r/w	r/o
$K = 1$	—	r/o	r/w	r/o

4. STORAGE AND PROCESSES IN CPR

Processes are separate units of program execution that do all the work in CPR, except for the limited functions in interrupt handlers of the supervisor and its extensions. The address space of each process is a subset of the segments in the single virtual address space of Figure 2 and is not limited to 16 segments.

4.1 Working (Short Address) Storage

Segment registers and a long virtual address are in the architecture to achieve adequate total virtual space. Use of short (32-bit) addressing where possible, however, remains very attractive. Space and time are saved if many storage references are to data and programs whose short addresses do not change. For these references, only short addresses need be remembered or passed in linkage, and no segment registers need be changed.

CPR uses short addressing for working storage, that is, active programs and temporary data of each process that exist only for the life of the process, for two reasons: First, though results are application dependent, measurements indicate that references to working storage greatly predominate over references to file data. Second, we think that the working storage and *protection domains* required by most processes can be contained in a few 256 Mbyte segments.

Protection domains are subsets of working storage with accessibility limited to subsets of all programs executing in a process. CPR provides three domains in each process: The supervisor accesses all working storage, the supervisor extensions (e.g., device drivers) access all working storage except that limited to the supervisor, and other programs access only working storage of the application domain. A few more domains could be added for subsystems, for example, database management.

To achieve short addressing, CPR limits the working storage of each process to a few segments of the single virtual space (up to eight segments or 2 Gbytes in our prototype) and loads their segment ids into fixed segment registers whenever the process executes. Segments used primarily for the supervisor and its extensions are shared by all processes (the same segment ids and registers). The program loader copies programs to be executed from permanent files (created by compilers and binders) into working storage segments. Temporary data are allocated in various working storage segments according to the protection domain. A working storage segment can contain many loaded programs and temporary data allocations, a one-to-many relationship.

In regard to working storage (not files), CPR is very similar to many widely used systems with virtual storage, such as DEC VMS and IBM MVS. Program and data addressing, including program linkage and parameter passing, occurs in a virtual space addressed by the 32-bit arithmetic of the processor. Each process has its own virtual space, divided into a few protection domains, with some domains shared by all processes. By contrast, Multics keeps each executing program and each separately allocated temporary data area in a separate segment of virtual space. The cost of this generality is that even nonfile addressing may fetch indirect address words from storage, sometimes cascaded, and may require segment register change. This is particularly evident in the Multics procedure call and parameter mechanisms described in [6].

4.2 File (Long Address) Storage

Files have multilevel names that are stored in a tree of file directories, as in Multics and now common in many systems. Directories contain information about each file, including a pointer to an *external page table* that contains the disk addresses of the file pages. When a file is first *opened* (a supervisor service), the authority of the calling process to open the file is verified. A segment of virtual storage (segment id) is allocated and associated with the file's external page table. The storage manager later services page faults in the segment by reading pages of the file. If a file is already open when *open* is called, and authority and compatibility with previous opens are verified, then the segment id already allocated is used to share the file. The relationship of segments to open files is one-to-one.

Because only the supervisor can load segment registers, the segment ids of files are kept in supervisor-protected tables. A small storage block, called a *refp* (reference to persistent), is used to represent a file segment id. An empty *refp* is passed in the call to open a file. *Open* stores the index of a supervisor table entry in the *refp*, and stores the address of the *refp* and the allocated segment id in the supervisor table entry. The *refp* is later passed in a supervisor call to *load segment register*. The supervisor verifies that the caller can access the storage of the *refp* and that the index in the *refp* was stored at that address by *open*. Thus assured that the *refp* is not forged, the supervisor loads the file segment id into a caller-specified segment register used for files (one of up to six in our prototype).

While a file segment id is in a segment register, a short address [*sreg*, offset] can access any byte in the file. Since a process may open many files and change the contents of segment registers to address them, however, programs must in general use a form of long address for file data [*refp*, offset].

*Refp*s, which are capabilities to segment ids, are in working storage and subject to the protection domains previously described. If files are opened by the supervisor (directory files are a good example) using *refp*s not in the application domain, then ordinary programs cannot access the *refp*s or the files. Thus, the protection of working storage domains extends to enforce file authorization by domain.

The CPR treatment of files in virtual storage is essentially equivalent to Multics. Files have symbolic names and lifetimes independent of virtual address. When a symbolic file name is referenced, a virtual segment is allocated, and the entire file may then be accessed in it. Multics claims no distinction between file and virtual segment, but its *make-known* operation allocates virtual storage to files just as does *open* in CPR. In System/38, however, each file is assigned virtual storage when first created and retains the same address for its entire lifetime, even if no process is using that address.

4.3 Process Address Space

A process address space consists of a few working storage segments, whose ids are always in registers, and file segments, whose ids (represented by opened *refp*s) may be loaded into registers. There are two language approaches to programming in this space, described in more detail later: First, given declarative language extensions, a compiler can manage the details of file segment register loading.

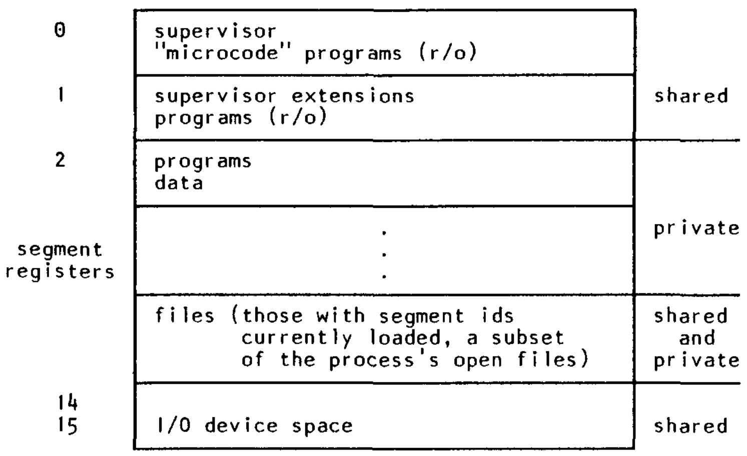
0
1
2
segment registers
14
15
supervisor "microcode" programs (r/o)
supervisor extensions programs (r/o)
programs data
files (those with segment ids currently loaded, a subset of the process's open files)
I/O device space
shared
private
shared and private
shared

Fig. 4. The short address space of one process.

Programs are mostly independent of variable location, file segment or not. Second, programs can explicitly manage file segment registers and use languages having pointers to address variables in file segments.

The short address space of a process is determined by the contents of segment registers, as shown in Figure 4 for our CPR prototype. Working storage segments for the supervisor and extensions are in registers 0 and 1, shared by all processes. A private working storage segment for the application domain is in register 2. Additionally, there are five registers reserved for future implementation (more working storage, more protection domains for subsystems, e.g., database management), six registers used for file segment ids, and two registers for hardware I/O space.

To improve program sharing, as in Multics, CPR permits a read-only program section (instructions and constants) to be loaded separately from the read-write section (data). CPR shares read-only library subroutines (string move, copy, etc., the RISC analog of microcode) and read-only sections of other frequently used programs by loading them in the segments of registers 0 and 1 (Figure 4). Read-write sections are loaded in the segment of register 2, along with nonshared programs.

The three protection domains mentioned previously do not correspond exactly to segment registers 0, 1, and 2. The read-only shared programs addressed by registers 0 and 1 are part of the application domain. Also, there is a small amount of data in the segment of register 2, private to each process, that is limited to the supervisor and extensions domains. These domains are enforced by the previously described page protection hardware; that is, each page is given a *PP* state, and segment register *K* bits are changed by the supervisor on call/return between domains.

It is important to note that there is no individual limit on the number of segments in the address space of each process. If a process requires more working storage than is short-addressable with eight segment registers or if it wants to share working storage with a subset of other processes, then it may use temporary file segments. The single virtual space of Figure 2 is a limit on the sum of all working storage and open file segments of all processes. If the single virtual space were extended (wider segment id and registers), this systemwide limit would increase to accommodate more processes, etc. Only storage manager programs would be affected by such an extension. All other programs use segment ids only indirectly through the *refp* mechanism, which can accommodate an essentially unlimited number of file segments in each process.

4.4 Transactions and Database Storage

A transaction in CPR is defined as all the storage actions by a process on a set of file segments, performed between two calls to *commit* (a supervisor service) for that set of segments. A transaction includes any storage actions of the supervisor, in directories or external page table segments, taken because committed segments are created, erased, renamed, enlarged, etc. Each process has a unique transaction id, which the dispatcher loads into the transaction id register when the process runs, to allow hardware to identify storage actions of the process in file segments for which transaction locking is requested. This one-to-one process/transaction relationship was chosen only to simplify the CPR prototype. In a system with nested or distributed transactions [15], the hardware id per process could be used to identify actions of subtransactions that were allowed to proceed asynchronously. Storage manager software would understand the relationship of processes to transactions, implement lock inheritance and nested commit, etc.

It is important that the transaction functions that CPR performs (locking, logging, etc.) are invoked implicitly, as a side effect of access to file segments by normal storage instructions. This allows CPR to assume correctness responsibilities (e.g., that objects are locked when referenced, that all updates are logged). If locking and logging were CPR services, called explicitly, then database managers would retain the major burden of transaction correctness, and little would be simplified. Our first idea for implicit transaction functions was that compilers could generate calls when data were referenced, based on declared attributes. However, this could be difficult to implement in many languages, performance might be poor, and subprograms would have to know if parameters were database or not. This led us to a hardware assist.

By database storage we mean file segments opened with options that cause the supervisor to assume responsibility for access serializability and/or atomic update in those segments. The user of database storage has two responsibilities: to use appropriate *open options* and to end each transaction by calling *commit. Undo* may instead be called, to back out any updates made by the transaction since the previous commit. A database manager or application in one process may otherwise access shared files without any explicit interaction with other sharing processes.

4.5 File Open Options

We first describe the options separately. *Read* or *write* (which also allows read) states the access intention of the opening process. *Read-write share* permits other processes to open the same file for writing or reading. Without it, sharing is limited to read-only. *Locks* requests access serializability of both writes and reads.

Journal requests write serializability and atomic update. *Replace* requests writing a new file to atomically replace the old. *Temporary* creates a file that exists only while open. We do not discuss authorization, that is, the right to open a particular file with certain options. We do describe some possible combinations of open options and the resulting transaction properties. The following four combinations of options provide the strongest properties of serializability and atomic update:

(1) The combination of *read* or *write* with *read-write share, locks, journal* requests the most general database storage. Other processes may open the file with these options at the same time. The transaction locking mechanism is used to provide serializability of access and atomic commit of updates by each process.
(2) *Write, journal* requests atomic commit of updates by one process. Other processes are prevented from opening the file. Transaction locking is used only to detect updates.
(3) *Write, replace* causes creation of a new file by one process. Other processes are prevented from opening the file. Transaction locking is not used in the file itself, but the file is written to disk and the directory is updated atomically to point to the new file, replacing the old if it exists.
(4) *Read,* used alone, allows other processes to open the file, but only with *read* alone. Transaction locking is not used, but all stores are prevented.

Weaker properties are provided by *read* or *write* with *read-write share, journal.* Other processes may open with these options. Transaction locking is used to serialize and atomically commit the writes of each process; but reads are unconstrained, that is, not serialized with writes. Use of this combination for file directories is described later.

Weakest is *write* of an existing file (without *replace* or *journal*) with or without *read-write share.* File updates may occur piecemeal prior to commit, and there is no access serializability.

Temporary, write provides a file segment that exists only until *closed.* A file name is used to open, but is not recorded in the directory. With *read-write share,* other processes may open the same name and share the storage. With *locks,* transaction locking is used to provide access serializability, and *commit* only releases locks.

5. PROGRAMMING LANGUAGE INTERFACE

To realize the benefits of files in virtual storage, the main requirement for a programming language is to allow the same data types, structures, and computations in files as in working storage. Also, the compiler should handle all addressing details, to simplify programming and avoid errors. However, to enable the compiler to handle these details and the differences between short and long addressing, a new storage class, *persistent,* and a new type, *refp* (*reference to persistent*), are added to PL.8.

Variables declared to be any of the usual data types and structures may also be declared to be of persistent storage class. A persistent variable must be associated with a variable of type *refp.* This *refp* represents the storage (segment

```
debit_credit: proc;

dcl ac_ref refp;
dcl 1 accounts(100000) persistent(ac_ref),
       2 code integer,
       2 balance integer,
       2 branch integer,
       2 other_info char(116);
/* other declarations */

$open(ac_ref, '/801_bank/accounts_file', dbase_options);
/* other initialization */

/* transaction processing main loop */
do while(tellers_active);
    /* get a transaction: account number, amount, teller */
    do until(rc ¬= deadlock);
        call do_trans(account, amount, teller) exception(rc);
        if rc = deadlock then $undo;
    end do;
end do;

/* procedure to perform one transaction */
do_trans: proc(ac, amt, tel);
  /* verify account number, sufficient teller balance, etc. */
  if balance(ac) + amt >= 0 then
     balance(ac) = balance(ac) + amt;
  /* update teller, branch, transaction audit trail files */
  $commit;
end do_trans;

end debit_credit;
```

Fig. 5. The debit-credit program.

id) that contains a file that has been opened by calling the supervisor. Though a *refp* could be implied with a persistent variable, *refp*s are made explicit in the language for generality with aggregates of files (details beyond this paper's scope). Semantics of computation with persistent variables, depend only on type and are the same as with the working storage classes automatic (stack), static, and controlled (heap).

5.1 Debit-Credit Example

Further details are best explained by an example shown in Figure 5. This program has been abstracted from a debit-credit application that we use for performance evaluation.

In the debit-credit program, each account record is an element in a persistent array *accounts* whose segment id is associated with *ac_ref*. The program calls supervisor service *$open* to find the actual accounts file and allocate a segment for it. The segment id is saved in the supervisor, and an index to it is stored in *ac_ref*. When the program needs to address a variable in *accounts*, the compiler generates a call to the supervisor, with *ac_ref* as one argument and register number as the other, to load the segment id into one of the segment registers used for files (Figure 4). The compiler allocates these registers (six in our prototype) and optimizes loading of them, as with other address computation [3].

The procedure *do_trans* at the end of the example performs one transaction. The computation shown is independent of storage class, except that the compiler uses long address references to *balance*, which read from and update directly in the accounts file. After similar updates to other files (not shown), the program calls supervisor service *$commit*. The program uses database options (previously described) in calls to *$open*, so lock fault interrupts occur as files are referenced, and there is associated processing at commit time, described later.

If deadlock occurs in this program, it is when *do_trans* references one of the shared files. The lock fault handler signals deadlock exception to a process chosen as victim. The PL.8 exception mechanism terminates the current procedure and looks in the call chain for an exception handler, indicated by an *exception* qualifier on the call statement. The example main loop handles exception return from *do_trans*, calls supervisor service *$undo* to back out partial updates when there is a deadlock, and repeats the call to *do_trans* until there is no deadlock.

5.2 Other Language Issues

PL.8 has other types more appropriate for database structures than the simple *accounts* array of the example. An entire file may be declared as an area of bytes of persistent storage class. Then, various structures may be allocated within this area to hold the records, index entries, etc., of a database. These structures may be intermixed and may contain offset pointers to form trees, etc. The program must allocate storage within the area, but the compiler loads segment registers as required. The persistent class also allows computation on very large aggregate types, exceeding 32-bit addressability, in storage provided by sets of files. Multidimensional arrays of structures or based structures in areas may be used. Generalized pointers are pairs [file, offset] where file identifies one of the set. The compiler handles addressing details.

Persistent variables may be passed in procedure calls. However, the compiler cannot load segment registers for them in the caller and pass short addresses, because the called procedure might use the same registers for its own persistent variables. Therefore, any parameters of a called procedure, for which a caller might pass persistent arguments, must be declared persistent. Such parameters are passed by long address [address of *refp*, offset in segment], and the called procedure loads the segment registers.

It is possible to write storage-class-generic procedures because a persistent parameter will accept a nonpersistent argument. In the caller, the compiler constructs a *refp* to represent the working storage segment that contains the argument, and then passes the *refp* and the offset in the segment of the argument. When this constructed *refp* is used by the called procedure to load a file segment register (Figure 4), the supervisor interprets it as a request to copy the segment id from a working storage segment register. This generality is overkill for many leaf and near-leaf procedures that do not declare any persistent variables. A declared procedure attribute could be added to inform the compiler. Persistent arguments to these procedures could then be passed by short address, after loading segment registers for them in the caller.

Note that no special language is required to achieve procedures that are independent of the open options of persistent variables passed to them. This is because transaction locking and logging are invoked by the hardware assist and require no programmer or compiler actions. By comparison, the language approach in Camelot [21] includes a library of C routines that invoke primitive system transaction functions, to simplify programming (lock, read, modify) with recoverable objects. However, a subprogram must know to use these routines if operating on a passed recoverable object. Alternatively, the calling program can make the proper calls, providing it knows what the subprogram will do or did.

Multics did not need the language additions we have described. Its compilers could treat addressing details for files, programs, and temporary data areas similarly because they each occupied separate virtual segments. We have already compared this approach with the short/long address distinction made in CPR.

Languages like Pascal or C (and assembler) may address file storage even without extension. *Refp*s may be constructed from existing structure types and passed to *open* as in PL.8. The program must allocate file segment registers (Figure 4), call the supervisor to load them, and then use 32-bit pointers that correspond to the loaded registers to address variables declared to represent the files. Other functions, including transactions, are as described for PL.8.

6. STORAGE MANAGER IMPLEMENTATION

6.1 Virtual Storage

The CPR storage manager is a single implementation (using address translation, real storage, and disk storage) that provides virtual working and file (including database) storage. In addition to the inverted page table, two important data structures are the *external page tables* and the *disk allocation maps.*

An external page table provides a mapping of a segment's virtual pages to disk. Each table entry contains the disk address and other status bits for a page. Because segments may grow large or be sparsely used, an external page table is, in general, an unbalanced tree of pages. The tree contains only pages needed to map virtual pages of the segment actually used. For file segments that use only a few pages, the disk addresses are kept in the file directory, and there is no external page table.

External page tables are themselves stored in segments. Tables can be large and are therefore made pageable. To prevent infinite recursion in page fault handling, there is special treatment of the table for a segment that contains other external page tables. This table is allocated in the segment that it maps, within a limited range of pages, and the pages that map that range are fixed in real storage. The recursion is programmed by backtracking, described later. Tables for file segments are kept in file segments and updated atomically, along with directories and the actual files, as described later.

There are disk allocation maps for both temporary disk space and permanent file spaces, with 1 bit per disk block. In general, disk space is allocated as pages are referenced. To improve sequential performance, a small group of consecutive blocks is reserved for a file and then allocated to it as needed. More groups are used as needed. The unused portion of a group is freed at file close.

Pages are generally written to the disk addresses allocated and stored in the external page tables, the *home* location. This is constrained for existing pages in file segments opened with the *journal* option. These pages are only written to their home locations if their contents have been committed. If the storage manager insists on paging out such a page containing uncommitted data, it chooses a temporary disk address, saves it in a table, and writes the page there until commit. This constraint is a consequence of logging only afterimages of changes, as described later, but is not inherent in database storage. New (never committed) pages may be written to home.

6.2 Backtracking and Careful Update

Most data used by the storage manager are shared by all processes and must therefore be referenced in *critical sections* to maintain consistency. Backtracking and careful update are used in critical sections that could page fault when referencing external page tables or disk allocation maps. On entry to such a critical section, a *backtrack state* is saved, which is the current process registers at that time. This state implicitly includes the request (page fault, paging service, etc.) that caused entry to the critical section. Careful update means that shared data are not updated until all pages needed to complete the update are first touched without causing page fault. Since the critical section prevents page stealing, the update can then be successfully completed.

If a page fault occurs, all computations since the backtrack state was saved (registers and automatic variables) are discarded, and the new page fault is handled. If paging I/O is necessary, the process leaves the critical section, and other processes may enter it, since shared data are consistent. When I/O finishes, the process reenters the critical section at the backtrack state to handle the original request from the beginning.

To complete such a critical section without incurring a page fault, a few pages must be in real storage simultaneously (e.g., up to three pages of external page tables). This requirement is the same as that for atomic execution of one complex machine instruction (e.g., up to eight pages for Move Characters in IBM System/370). Experience indicates that demand paging (one page-in at a time, no page fixing) eventually meets this requirement without more formal guarantees. The only alternatives to backtracking and careful update that we know of are to page fix storage manager tables in real storage or to handle storage manager page faults serially with special logic.

6.3 Locking

Implementation of locking is based on the architecture, transaction definition, and open options previously described. Each transaction (process) has a unique id stored in the transaction id register when the process runs. If a file is opened with the *locks* or *journal* options, locking is activated by setting segment register *S* bit (Figure 1) for the file segment. As transactions access this segment, the hardware causes lock fault interrupt whenever locks in the inverted page table entry for a page (Figure 3) are not those of the current transaction or do not allow a specific load or store instruction. The storage manager of CPR searches a *lock table*, makes the transaction wait if there are conflicting locks, or grants and records the requested lock in the table. The manager then copies all locks of the current transaction in the referenced page to the inverted page table entry, so that the program that encountered lock fault may proceed.

Locks are granted to and held by transactions until they commit: share for read and exclusive for write. It should be noted that such locking of arbitrary storage units (128 bytes) achieves transaction serializability [8] independently of the data structures and algorithms in programs that use the storage. If a file is opened with the *journal* option, but not the *locks* option, then only write locks are recorded in the lock table and used for conflict analysis. Read locks are granted freely and not recorded.

The lock table contains entries called *lockwords*. Each lockword has fields similar to those of inverted page table entries (Figure 3), that is, *segment id*, *vpage*, *tid*, *W*, and *lockbits*. A lockword is allocated when a transaction first references a page following commit and is freed by commit. There is an allocated lockword for each page referenced by each transaction since its last commit. Each lockword is also on two lists: one a hash of *segment id* and *vpage*; the other by *tid*. Lock fault processing uses the hash to find all locks of any transaction in the referenced page, to detect conflicts and record locks granted or requested. Commit processing uses the transaction list to find all locks held by the committing transaction in all pages. The lock table is pageable and is referenced in storage manager critical sections using backtracking and careful update, as previously described.

6.4 Commit

A call to *commit* atomically applies all updates of a transaction to three types of data: actual files to be committed, external page tables of those files (if disk blocks were allocated), and related file directories (for new files, erase, rename). Because we want to avoid deadlock in *commit* and other supervisor services, the external page tables and directories are treated specially. Their segments are opened with the previously described combination of options that provides hardware locking for write serializability and commit, but no read locking. To achieve full consistency, conventional software locks are acquired, in a deadlock-free order, before access to these two types of data in open, commit, etc., and are then released. Also, these data are updated only during commit, so the duration of write locking is very short.

The log is written sequentially, in disk space separate from files. Since all three types of data to be committed are in segments that use *lockbits*, logging is simply copying to the log each 128-byte *line* of storage for which the transaction holds a write lock. An *end-transaction* record is added, the log is forced to disk, and all the transaction's locks are released. This log force is the atomic commit. If the end-transaction record is not in the log after a failure, the transaction's log records are discarded. Since uncommitted pages are not written to home disk locations, the files remain unchanged. The choice of only afterimage logging and not writing uncommitted pages is not required to implement database storage.

The architecture allows equally well the additional logging of beforeimages, and we could then write uncommitted pages to home. We made the choice to achieve a shorter log (approximately half) because of the technology trend to ever-larger real storage that can accommodate more uncommitted pages until commit.

Undo is implemented by discarding uncommitted pages from real storage or from temporary disk, if any were written there. The previously committed versions of pages are reinstated. This is possible because, as described earlier, the lockbits only allow one writer in a page, we do not write uncommitted pages to home disk locations, and we chose to write committed pages to home disk immediately after commit. Frequently changed "hot" pages require special treatment to reduce disk writes. If a page is referenced while awaiting disk write, it causes a page fault, and a copy of the page is made for the new requester. If the new copy is updated and committed before the first write is performed, the new copy simply replaces the old on the disk write queue. Although updates in hot pages are always written to the log by commit, the pages themselves are not written for every commit. The first reference to a page after undo gets the latest committed version, either in real storage or from home disk. Periodic checkpoints force hot pages to disk, to limit the work of future redo.

Redo from the log after system failure is accomplished by copying into permanent file pages all afterimage log records of recently committed (since the last checkpoint) transactions. This is *idempotent* and therefore restartable if failure occurs during redo. Our CPR prototype does not implement recovery from disk media failures, for example, head crash or, with some disks, power loss during write. An archive of the log would provide backup sufficient for media recovery, since the log contains all data stored in logged files. The log could also be used for an alternative implementation of undo, which resembles redo. If committed pages were not immediately written to disk and subsequent transactions were allowed to update such pages without first making a copy, then, if undo were necessary, the committed versions could be reconstructed from home disk and recently committed afterimages.

7. PERFORMANCE AND PROTOTYPE EXPERIENCE

We first consider the cost and performance of the base storage facilities themselves. The RT PC storage controller (MMU) is one chip of NMOS logic, including the address translation and protection facilities [12]. Less than 10 percent of the area of this chip is devoted to the transaction locking mechanism. The storage manager of the CPR supervisor is less than 15,000 lines of mostly PL.8 programming, including file directories, simple access methods, virtual storage, real storage, disk storage, page and lock fault handlers, logging, open, commit, undo, erase, rename, etc.

Approximate instruction counts of some supervisor functions that might be compared with analogous functions performed in a conventional buffer pool manager are given in Table I. The 1400 instructions for a page fault include about 350 instructions for a very simple disk driver in a one-disk system. It also includes returning to the process dispatcher because of the page fault and being redispatched when the disk read is complete. Locks associated with database storage are granted by an interrupt handler and, when there is no conflict for

Table I. Supervisor Path Lengths

Function	Instructions
Page fault with disk read	1400
No-conflict lock grant	250
Load segment register SVC	25

Table II. Debit–Credit File Sizes

File	Array size	Size in kbytes
Accounts	15,000	4,000
Tellers	200	20
Branches	8	4
History	10,000	500

Table III. Debit–Credit Performance

System	Instructions (in thousands)	Disk I/O
Lean and mean	20	6
Fast	50	4
Good	100	10
Common	300	20
CPR on RT PC	22+	6+

the lock, the interrupted process continues execution after a cost of about 250 instructions. A supervisor call (SVC) is required to load a segment register because it is a privileged instruction and because it is necessary to verify authorization, as previously described. Because segment registers can be frequently loaded, this SVC was specially coded to avoid the overhead of CPR's general SVC handler.

We next give some performance figures for the debit-credit application that has been used to benchmark the performance of on-line transaction systems [7]. The database component of this transaction consists of debiting a bank account, updating a teller and branch balance, and writing to a history file a record of the transaction. In debit-credit we use four persistent variables: accounts, tellers, branches, and history. Each is declared as an array of structures as in the previous PL.8 example. Table II gives the actual sizes of the four files that could conveniently fit into our hardware, which has a single 37-Mbyte disk and special hardware for counting instructions executed.

In an actual system, the branch and teller files are larger, but would still easily fit in real storage, whereas the accounts file in our example is larger than real storage. Consequently, in considering disk I/O, our example reasonably reflects the performance that would be experienced in a more realistic system. Table III gives the path lengths and disk I/O counts measured in CPR on the RT PC and for several other systems that have been reported [7].

The path length for CPR does not include the X.25 block terminal support that is included in the other systems. Also, the log is not duplexed in CPR; if it were, the disk I/O count would have been approximately 1 greater. The

Table IV. A Subset of the DeWitt Benchmarks (elasped time in seconds)

Benchmark	SQL/RT	SQL/801	SQL/801 No locks
1 percent select (unique)	2.3	2.1	2.0
10 percent select (unique)	11.2	19.1	18.0
1 percent select (secondary)	6.5	4.1	3.9
10 percent select (secondary)	48.8	28.9	28.0
JoinAselB	29.0	61.0	59.5
JoinABprime	31.2	85.0	83.0
JoinCselAselB	38.6	36.0	35.0
sJoinAselB	120.0	84.5	78.0
sJoinABprime	43.0	85.0	81.0
sJoinCselAselB	141.4	57.0	55.5
Insert one row	1.5	0.6	—
Delete one row	1.4	0.5	—

figures given for CPR are averages for a range of levels of multiprogramming from 1 to 10. Deadlocks were a few percent at worst. Because of the limited disk configuration, however, no conclusions can be drawn about concurrency issues from this example. Indeed, these performance figures are only given to indicate that the cost of using database storage is not obviously too high to be of practical use.

Attanasio [2] describes experiences with implementing SQL/801, a program that supports a large subset of the relational database language SQL (most verbs, views, *n*-way joins, some optimization, excluding subqueries and aggregate functions). The development of SQL/801 was greatly simplified because, with minor exceptions, it considers only a single user. It achieves multiuser concurrency by running in multiple processes using the shared database storage of CPR, and is coded in less than 15,000 lines of PL.8. Table IV gives some performance figures for SQL/RT and SQL/801 on a subset of the DeWitt benchmarks [5]. SQL/RT is a product, implemented with conventional database techniques, supporting SQL and running on the AIX operating system version 2.1.

The timings in Table IV were all obtained on the same hardware configuration: an RT PC 6150 model 025 with 4 Mbytes of real storage and a 70-Mbyte disk. SQL/801 No Locks is the same as SQL/801, except that locking was turned off by opening all the segments without the *lock* and *journal* options. The difference between them is a measure of hardware and software locking cost. The select queries retrieve from relations having 10,000 tuples, each tuple having 16 fields and a length of 182 bytes. The join queries operate on the same relations or smaller ones with 1,000 tuples. Insert and delete involve the same 10,000-tuple relations that have one primary (unique) index and two secondary indices. The indices are on fields with integer attributes. Result tuples are inserted into a relation. Times shown are the wall clock time in seconds. Again, these figures are reported only to indicate that the cost of database storage is not obviously too high.

8. SUMMARY AND CONCLUSIONS

We have discussed 801 storage architecture, currently implemented in the IBM RT PC hardware, and a storage manager and language interface, which are parts of the CPR prototype operating system. Together, they offer large virtual storage, access to files in storage with sharing, and the innovation of database storage, which provides serializable atomic transactions on files, implicitly as storage is used.

Short (32-bit) addressing is chosen for most computation and program linkage. Longer virtual addresses are used for files and unusually large computations. This addressing distinction is efficient for the most frequent storage references and allows compatible extension of the addressing range, from small to very large machines.

A novel hardware locking mechanism monitors the read and write references of individual transactions to 128-byte lines of database storage. Lock fault interrupts invoke the storage manager to grant locks, and later, when a transaction commits, the storage manager writes log records of changed storage. This achieves correct serializability and atomic update in linear byte files, without explicit calls from programs that access storage and independently of superimposed data organization or access pattern.

The PL.8 programming interface to storage provides uniform access to an essentially unlimited number of variables, of any type and structure, some of which may be files. The language hides most distinctions between short and long addressing, except for declaration of storage class. The compiler manages addressing details, including loading of segment registers when necessary.

Experience with our prototype indicates that these storage facilities can greatly simplify the programming of database subsystems and applications, and offer good performance for the workloads studied thus far. Implementation cost is modest for hardware and software. More work is needed, with adequate I/O configurations and workload variety, to explore properly the limits and parameters of performance.

References [22]–[25] have raised issues about database management with virtual storage and/or operating-system transactions, for example, that LRU replacement and entry-per-page mappings of virtual storage may not be best for files and that coordination of components may be required to prevent writing virtual pages to disk until log records are written. CPR addresses these issues by providing paging hint calls and by integrating the implementation of storage and logging. Compact mappings of files to extents could certainly be added. These same references have also noted that locking based on units of storage accessed seems incompatible with techniques used in some database systems to improve performance and concurrency; for example, locking and logging only data changes (effects on indices implied), and releasing some locks before commit. Similarly, database storage prevents operations like increment/decrement from proceeding without locks, as is done in IMS Fast Path. These topics and their importance in practical performance should be studied.

Database storage is a new way to implement certain storage management functions in an operating system, built on and similar in spirit to virtual storage. Both are very general, transparent, and rather monolithic approaches to storage management—one for storage hierarchy, and the other for storage concurrency and recovery. We believe that database storage will perform well for a wide range of applications and that the simplicity it offers is too attractive to dismiss. As in the early days of virtual storage, the challenge is to understand and exploit its characteristics.

ACKNOWLEDGMENTS

John Cocke and George Radin made original contributions to the ideas presented. Many other members of the 801 project have also contributed to design and implementation. We particularly acknowledge Dick Attanasio, who implemented SQL/801 and made other contributions to performance evaluation. Important help with this paper was given by Michael Blasgen, Dick Attanasio, Marc Auslander, Charles Barton, Darlene Bennett, Tony Cocchi, Kati Rader, and several reports from referees. John Bissell supplied the SQL/RT performance figures.

REFERENCES

1. ASTRAHAN, M. M., BLASGEN, M. W., CHAMBERLIN, D. D., ESWARAN, K. P., GRAY, J. N., GRIFFITHS, P. P., KING, W. F., LORIE, R. A., MCJONES, P. R., MEHL, J. W., PUTZOLU, G. R., TRAIGER, I. L., WADE, B. W., AND WATSON, V. System R: A relational approach to database management. *ACM Trans. Database Syst. 1*, 2 (June 1976), 97–137.
2. ATTANASIO, C. R. 801 architecture support for database—A case study. Rep. RC12416, IBM T. J. Watson Research Center, Yorktown Heights, N.Y., 1987.
3. AUSLANDER, M., AND HOPKINS, M. An overview of the PL.8 compiler. SIGPLAN 82 Symposium on Compiler Construction. *ACM SIGPLAN Not. 17*, 6 (June 1982), 22–31.
4. BENSOUSSAN, A., CLINGEN, C. I., AND DALEY, R. C. The Multics virtual memory: Concepts and design. *Commun. ACM 15*, 5 (May 1972), 308–318.
5. BITTON, D., DEWITT, D. J., AND TURBYFILL, C. Benchmarking database systems: A systematic approach. In *Proceedings of the 9th International Conference on Very Large Data Bases* (Florence, Nov. 1983). VLDB Endowment, Saratoga, Calif., 1983, 8–19.
6. DALEY, R. C., AND DENNIS, J. B. Virtual memory, processes, and sharing in MULTICS. *Commun. ACM 11*, 5 (May 1968), 306–312.
7. DATAMATION. A measure of transaction processing power. *Datamation 31*, 7 (Apr. 1, 1985), 112–118.
8. GRAY, J. N. Notes on database operating systems. In *Operating Systems—An Advanced Course*, R. Bayer, R. M. Graham, G. Seegmuller, Eds. Springer-Verlag, New York, 1978, 393–481.
9. GRAY, J. N. The transaction concept: Virtues and limitations. In *Proceedings of the 7th International Conference on Very Large Data Bases* (Cannes, Sept. 1981). IEEE, New York, 1981, 144–154.
10. IBM. IBM System/38 technical developments. Order no. G580-0237, IBM, Atlanta, Ga., 1978.
11. IBM. IBM RT PC hardware technical reference. Order no. SV21-8024, IBM, Austin, Tex., 1985.
12. IBM. IBM RT personal computer technology. Order no. SA23-1057, IBM, Austin, Tex., 1986.
13. LAUER, H. C. Observations on the development of an operating system. *Oper. Syst. Rev. 15*, 5 (Dec. 1981), 30–36.
14. MCGEE, W. C. The information management system IMS/VS. *IBM Syst. J. 16*, 2 (1977), 84–168.
15. MOSS, J. E. B. Nested transactions and reliable distributed computing. In *Proceedings of the 2nd Symposium on Reliability in Distributed Software and Database Systems* (Pittsburgh, Pa., July 1982). ACM, New York and IEEE, New York, 1982, 33–39.
16. PATTERSON, D. A. Reduced instruction set computers. *Commun. ACM 28*, 1 (Jan. 1985), 8–21.
17. RADIN, G. The 801 minicomputer. *SIGPLAN Not. 17*, 4 (Apr. 1982), 39–47.
18. REDELL, D. D., DALAL, Y. K., HORSLEY, T. R., LAUER, H. C., LYNCH, W. C., MCJONES, P. R., MURRAY, H. G., AND PURCELL, S. C. Pilot: An operating system for a personal computer. *Commun. ACM 23*, 2 (Feb. 1980), 81–92.
19. SIMPSON, R. O. The IBM RT personal computer. *Byte 11*, 11 (1986), 43–78.

20. SMITH, A. J. Cache memories. *ACM Comput. Surv. 14*, 3 (Sept. 1982), 473–530.
21. SPECTOR, A. Z. Distributed transaction processing and the Camelot system. In *Distributed Operating Systems—Theory and Practice*, Y. Paker, Ed. Springer-Verlag, New York, 1987, 331–353. (Also Rep. CMU-CS-87-100, Carnegie-Mellon Univ., Pittsburgh, Pa., 1987.)
22. STONEBRAKER, M. Operating system support for database management. *Commun. ACM 24*, 7 (July 1981), 412–418.
23. STONEBRAKER, M. Virtual memory transaction management. *Oper. Syst. Rev. 18*, 2 (Apr. 1984), 8–16.
24. STONEBRAKER, M., DUBOURDIEUX, D., AND EDWARDS, W. Problems in supporting database transactions in an operating system transaction manager. *Oper. Syst. Rev. 19*, 1 (Jan. 1985), 6–14.
25. TRAIGER, I. L. Virtual memory management for database systems. *Oper. Syst. Rev. 16*, 4 (Oct. 1982), 26–48.
26. WALKER, B., POPEK, G., ENGLISH, R., KLINE, R., AND THIEL, G. The LOCUS distributed operating system. *Oper. Syst. Rev. 17*, 5 (Oct. 1983), 49–70.

Received May 1987; revised August 1987; accepted September 1987