


requirements of VMDB systems while potentially allowing all the benefits of the VMDB approach to be realized.

We have examined the impact of the RPVM module on the performance of simple programs that use virtual memory and the new RPVM primitives, and found that our modifications do not impose significant virtual memory overhead. Our results indicate that the retrieval and processing of potentially constraining rules can potentially be a source of significant overhead. Further studies are needed to ascertain whether commonly used recovery techniques result in page-flush policies with flush-rule ratios close to zero and a large average number of potentially constraining rules per page and if so, how this problem can be addressed. Other future work includes comparing the performance of this system with a similar system implemented as an external pager, and studying how the kernel's page replacement policy and database system performance can be further improved by exploiting information present in the page-flush policies of the RPVM module.

References


7.5 RPVM Performance

From our analysis and experiments, it is clear that negligible additional computational overhead is experienced by existing applications that do not use the new RPVM primitives.

Paging operations involving pages of RPVM objects, however, do experience non-negligible additional overhead. The time taken to perform a page-in operation for a page of a RPVM memory object is dependent on the number of pages from that object that have been paged to the swap space. This in turn is dependent on the amount of main memory allocated to database pages and the percentage of database pages in main memory that have been modified and are not propagatable. Only if these two factors combine to result in a very large number of database pages being paged to the swap space and a significant increase in locate page step costs will any significant degradation in average page-in performance be observed.

The time taken to perform a page-out operation involving a page of a RPVM memory object is dependent on the size and nature of the page-flush policy corresponding to the memory object. The additional overhead incurred is particularly sensitive to the number of potentially constraining rules of the page, especially if the page's flush-rule ratio is close to zero. Although in principle, there can be as many such rules as there are pages in an address space or database, in practice this number is likely to be small. Moreover, a limit can be placed on the number of rules that are processed. If this threshold is reached, the page can be treated as if it is constrained.

8. Conclusions

Our initial work indicates that it is feasible to add direct and general support for Recoverable-Persistent Virtual Memory based on the RPU model to the Mach kernel. We were able to do so by building upon existing kernel data structures and functions in a modular way and by making small changes to the virtual memory page-in and page-out paths. The RPVM user-interface introduces only a few new system calls and abstractions. We believe that other operating systems that support memory-mapped files can similarly be extended to realize RPVM. The RPVM system adequately supports the consistency and recoverability
is subsequently required in main memory. The cost of this update operation should be comparable to the cost of the locate page step of the RPVM page-in operation. Except for this additional cost, the swap space pageout cost incurred should be no higher than if the selected page was sent to its external pager.

7.4 Page-out Performance

The same kind of measurements taken for page-in operations were also taken for page-out operations. The performance of a program that sequentially writes one byte of each mapped database page was measured. These modified pages are paged-out back to their external pager when page replacement occurs.

We measured the time taken by a program to write and the kernel to page-out modified RPVM pages when no propagation constraints apply. The average additional overhead incurred by the determine propagatability step was 169 microseconds or 3.4% of the average cost (4869.5 microseconds) of handling a single page fault. The measurements are shown in Figure 4. Since page-outs are invoked to make memory resources available for page fault processing, we compare the additional page-out cost to the average computational cost of processing a single page fault.

Our experiments indicate that if a small number of potentially constraining page-flush rules exists for the pages being paged-out, then the additional cost incurred by the determine propagatability step is not significant. In the situation where the flush-rule ratio is 0, that is there are no constraining rules, and there are 2 potentially constraining rules for each database page that is replaced, the average page fault increased by 262 microseconds or 5.4%. With 4 potentially constraining rules per page, the page-fault cost increased by 410 microseconds or 8.4%. These measurements are shown in Figure 5. The average marginal cost per additional potentially constraining rule is 74 microseconds or 1.5% of the page fault cost of a non-RPVM object. Thus, when the flush-rule ratio is close to 1 or when the average number of potentially constraining rules per RPVM page is not too large, we do not expect support for RPVM to significantly impact page-out and page fault performance.

![Graph showing marginal times for satisfying each additional page fault for non-RPVM and RPVM memory objects](image)

This graph shows the marginal times for satisfying each additional page fault for non-RPVM and RPVM memory objects, after page replacement has begun. The time taken by a program to write a given number of pages of the memory object was measured. The marginal page fault time consists of the marginal page-in cost as well as the amortized marginal cost of page-out operations. The page-fault times for the RPVM object are for the case where the pages replaced have zero potentially constraining rules and none of the pages of the object have been paged to the swap space.

**Figure 4: Measured Performance of RPVM Write Operations**
7.3 Page-out/Replacement Analysis

Page-out operations send pages to their respective pages, and these are typically written to nonvolatile storage. In the Mach kernel, this operation is invoked by the pageout daemon when modified pages resident in main memory are selected for replacement. In RPVM the page-out operation may perform the following additional steps:

- **Ascertain RPVM object.** Ascertain if the selected page belongs to a RPVM memory object. If the page is not from a RPVM object, the original pageout path is taken. Otherwise the next two steps may also be performed.
- **Determine Propagatability.** Determine if the page can be propagated. This involves retrieving all relevant flush rules and flush-locks that specify propagation constraints on that page and processing all rules that apply.
- **Swap space Page-out.** If the page cannot be propagated, page it to the swap space and record its location. Otherwise, the original pageout path is taken.

In kernel-based RPVM, all page-out operations perform the additional *ascertain RPVM object* step. As in the case of page-in, this negligible overhead is the only additional direct cost page-out operations that handle pages belonging to non-RPVM memory objects.

The *determine propagatability* step is performed if and only if the selected page has been modified and belongs to a RPVM memory object. We refer to a page-flush rule, say A \(\leq p\) B, as a potentially constraining rule of page B. A rule is a constraining rule of page B if it is a potentially constraining rule of page B, and page A is either flush-locked or has been modified. We use the term *policy size* to refer to the number of different propagation constraints (flush-locks and rules) that constitute a policy.

Depending on the data structures used, the cost of retrieving all potentially constraining rules of a selected page may be sensitive to the total number of rules present in its page-flush policy. In our prototype system, rules are indexed via a hash function. If an appropriate hash function is used and there are not too many pages that have rules specified, then this cost should be independent of the policy size. The same holds for accessing flush-locks. Thus, the cost of the *determine propagatability* step is unlikely to be affected by the policy size but is probably highly dependent on the existence and number of potentially constraining rules. For each RPVM page being paged-out, one of three different cost scenarios applies.

First, if no propagation constraints apply, then the selected page can be propagated and the cost of this step is the cost of determining that no flush-locks and potentially constraining rules exists for the selected page. This cost is independent of the policy size if the policy is not very large.

Second, if the selected page has potentially constraining rules but is not constrained, then it can also be propagated. In this case, the cost of the *determine propagatability* step is directly proportional to the number of such potentially constraining rules. For each potentially constraining rule that is found, the state of the corresponding potentially constraining page has to be obtained. Since none of the rules constrain the selected page’s propagation, all such rules have to be retrieved and processed in this way, unless we short-circuit the process by having a threshold. Thus, this processing is probably the greatest potential source of overhead.

Third, if the selected page has a constraining rule, and thus cannot be propagated, the cost of the *determine propagatability* step depends directly on the number of potentially constraining but unconstraining rules that are processed before a constraining rule is encountered.

To facilitate the estimation of the cost incurred by the *determine propagatability* step in the case where there are potentially constraining rules, we define the *flush-rule ratio* of a page to be the ratio of the total number of constraining rules to the total number of potentially constraining rules of the page. As the ratio approaches 0, the average cost of this step is increasingly dependent on the number of potentially constraining rules. On the other hand, as it approaches 1, the average cost of this step becomes increasingly independent of the number of such rules and approaches the cost of processing a single potentially constraining page-flush rule.

The *swap space page-out* step is only performed for pages that cannot be propagated and which have been selected for replacement. Before such a selected page can be paged to the swap space, the identity of the page and related information are inserted into the swap hash table so that the page can be located when it
page-in operations involving pages from a RPVM memory object, the same program was used except that memory-mapped database was made a RPVM memory object by adding some suitable propagation constraint on its pages. For each experiment, the times taken for different sample sizes were measured. We used the marginal time taken to perform a page-in operation to represent the cost of such an operation as it eliminates the effects of page-ins at the beginning of each run that did not involve page replacement. The marginal time for a sample size is computed by dividing the difference in the time measured for this sample size and the time for the next smaller sample size by the difference in sample sizes.

To ascertain the additional cost of the locate page step when the swap space does not contain pages of a RPVM memory object, we measured the average time taken by a program to read and page-in a page belonging to the object from its external pager. We compared this to the case where the memory object read is a non-RPVM object. The difference between the two measured times represents the cost of the locate page step. The measurements from this experiment are shown in Figure 2. The observed average difference is 45 microseconds per page read. Thus, the additional cost incurred by the locate page step is 1.15% of the read page-fault cost of a non-RPVM object in this scenario.

The swap hash table is not empty if some RPVM pages are in the swap space. However, if an appropriate swap hash function is used, then the additional cost of the locate page step will remain small. To verify this, we created a scenario where a large number of constrained pages from a RPVM memory object were replaced from main memory and paged into the swap space. Then we measured the average time taken by a program to read and page-in the remaining pages of the RPVM object from its external pager. We found that when the number of swapped pages is less than or equal to the number of buckets in the swap hash table, the additional overhead incurred by the locate page step was negligible when compared with the case when no pages were swapped. When the number of swapped pages increases to 10 times the number of buckets, the locate page step overhead is only 18.5 microseconds more than if the swap hash table is empty or 1.6% of the cost of a non-RPVM object page-in. When the number of pages increases to 20 times the number of hash buckets, the overhead increases to only 36 microseconds more than if the swap hash table is empty or 2% of the cost of a non-RPVM object page-in. The measurements from this experiment are shown in Figure 3. From this data, we conclude that implementing RPVM in the kernel has very little impact on the performance of page-in operations.

![Figure 3: Measured Performance of RPVM Page-in with Pages in the Swap Space](image-url)
In kernel-based RPVM, all page-in operations have to perform the ascertain RPVM object step that can involve only a single read of a RPVM field, followed by a compare instruction. This negligible overhead is the only additional direct cost to page-in operations that handle pages belonging to non-RPVM memory objects.

In the case of memory page-ins involving RPVM memory objects, the locate page step is also performed. The cost of this additional step is the cost of checking whether the required page has been paged to the swap space. Recall that in our prototype this information is stored in a hash table. If none of the pages of the memory object has been modified or there has been no page replacement involving its modified and constrained pages, then the hash table will be empty and the cost of this step should be small.

If the required page has been paged to the swap space, the swap space page-in step is performed -- the page is requested from the default memory manager instead of its external pager. The page is in the swap space only if the required page had previously been modified and selected for replacement. Preliminary results show that for this step to be as efficient as traditional page-in operations that fetch pages solely from external pagers, the default memory manager must support prefetch.

### 7.2 Page-in Performance

We conducted our experiments on a prototype RPVM operating system running on a Sun 3/60 workstation with 16 Mbytes of main memory. The prototype RPVM system was derived from the Mach 3.0 microkernel (version MK75) with a user-level Unix-server (version UX38). The system uses a virtual memory page size of 8 Kbytes. We used an external pager (the modified Unix-server pager) that was extended to satisfy kernel data requests and writes without performing real I/O or paging on behalf of the page-in and page-out operations being measured. This allowed us to use wallclock time to represent the computational time taken by the kernel to perform the paging operations. We report the measured times in microseconds per paging operation.

![Graph showing measured performance of RPVM page-in operations](image)

This graph shows the marginal times for each additional page-in operation for pages from non-RPVM and RPVM memory objects, after page replacement has begun. The time taken by a program to read a given number of pages of the memory object was measured. The page-in times for the RPVM object are for the case when zero pages of the object have been paged to the swap space.

**Figure 2: Measured Performance of Basic RPVM Page-in Operations**

To measure the overhead incurred by a page-in operation, we used a simple program that mapped a file, representing a database and managed by a user-level external pager, into the program’s address space. The program sequentially reads one byte of each page in a range of pages belonging the mapped database. The time taken by the program to perform these reads for a number of pages or sample size represents the time taken by the kernel to perform the same number of page-in operations. To measure the cost associated with
one of the key reasons for using the VMDB approach, using external pagers in this way to realize RPVM systems is unsatisfactory. If the RPVM module is implemented within the kernel, then double paging can still occur. However, double paging can be avoided if the RPVM module can avoid sending the page being considered for propagation to either the swap space or the database when the page has already been paged out to the swap space. An external pager would not be able to accurately detect such a condition given the existing pager interface and that it can be preempted.

We note that while it is possible to construct pathological examples, experiments will ultimately be required to determine whether these performance costs are significant in practice.

### 6.2 Placing RPVM in the Kernel

A kernel implementation of the RPVM module is superior to an external pager-based implementation for three key reasons. First, in a uniprocessor environment, the non-preemptibility of kernel threads allows the modification state of pages to be determined simply and with little overhead. Second, accurate information about the state of the machine, memory use, and the location and state of database pages can be accessed with little overhead. Third, constrained pages that are selected for replacement can be paged to the swap space explicitly. As a result, most of the problems mentioned above that apply to an external pager-based implementation do not apply to a kernel-based implementation.

In addition, by integrating the RPVM module with the kernel, the potential for improved kernel operations exists. For example, page replacement policies could be optimized with information maintained by the RPVM module. For example, if pages A and B have both been modified and the rule A ≤p B holds, the replacement policy may select page A together with page B, if both are on the inactive queue. If Mach provided support for user-level replacement policies [4], this would be a lesser concern. Prefetching of database pages from the swap space based on the semantic information found in P-FB relationships could also improve virtual memory performance. We also hope to show that the writing of modified pages back to their backing stores can be accomplished more efficiently with this approach.

### 7. Performance Evaluation

It is our hypothesis that kernel support for RPVM does not introduce significant additional overhead to existing virtual memory operations, and that it can also improve the overall performance of database programs. In this section, we provide evidence in support of the first part of our hypothesis.

Implementing RPVM in the kernel requires modifications that primarily extend the virtual memory page-in and page-out path lengths. Among virtual memory operations, the page-in and page-out operations are among the most frequently invoked. It follows that the computational costs incurred by these paging operations can have a significant effect on virtual memory and program performance. Hence, our initial evaluation efforts were centered on the impact of the RPVM modifications on these paging costs. Both analytical and initial experimental data suggest that a kernel-based RPVM does not in general significantly impact virtual memory performance.

### 7.1 Page-in Analysis

Virtual memory page-in operations, invoked as a result of page faults, fetch pages from their pagers, and these pages are typically read from nonvolatile storage. In RPVM, the following additional steps may performed by the page-in operation:

- **Ascertain RPVM object.** Ascertain if the page belongs to a RPVM memory object. If so, the next two operations may be necessary. Otherwise, the original page-in path is taken.
- **Locate page.** Ascertain if the required page is in the swap space. If not, the traditional page-in path is taken and the page is requested from its external pager.
- **Swap space page-in.** If the page is in the swap space, fetch it in from the default memory manager.
B such that $A \leq B$. These structures are also involved in the determine propagatability step of the page propagation procedure and pageout daemon.

The above data structures are allocated only when needed and deallocated when empty, and are allocated on a per-object basis. In fact, a memory object is a RPVM memory object if and only if either a Flushlock or RuleTable structure has been allocated for it. Thus, these structures are not allocated for non-RPVM memory objects. Also, if database pages are not paged to the swap space, the SwapHash Table is not allocated.

Many of the data structures were implemented in C++, building on the GNU class libraries [3]. The hope is that by using C++, replacing the data structures will prove relatively painless. Integrating C++ code into the kernel was straightforward, requiring only a slight modification to the memory allocation routines.

6 Kernel vs External Pager

It is apparent from the previous section that the RPVM module operates very much like an external pager. Indeed, in principle the RPU model can be implemented outside the kernel in a specialized external pager. However, we chose to base our work on a kernel-based RPVM system because of the limitations imposed by the external pager-kernel interface and the opportunities for improved kernel performance presented by direct kernel support.

6.1 Limitations of an External Pager-based RPVM

One difficulty with implementing the RPVM mechanisms in an external pager is that the external pager may have outdated information on which pages are clean or dirty. Consider the case where the RPVM module is implemented in an external pager and it has received a page, say B, from the kernel. Let there be a P-FB rule that indicates that page A potentially constrains page B. Page A has been provided to the kernel. Even though it may have been clean when it was last seen by the external pager, page A could have been modified in the kernel's cache. Thus, before it can propagate page B, the external pager needs to determine the modification state of page A. This information must be determined through an exchange of messages between it and the kernel involving the use of the functions `memory_object_lock_request()` on page A, and `memory_object_lock_completed()` or `memory_object_data_write()`. If page A is clean, the external pager may proceed to propagate it, provided there are no other constraints.

If page A is dirty and can be propagated, the external pager propagates it before proceeding with the propagation of B. However, if updates affecting page A and the above message exchange can occur concurrently, that is page A is allowed to be flush-locked and updated after the `memory_object_lock_request` on A is called and before a copy of page A from the kernel is received, then the copy of page A that is acquired by the pager may not be consistent. This problem is avoided if updates are blocked while the flush-lock status of page A is being determined and a copy of page A is acquired. Hence this problem does not arise in a RPVM that is implemented within a uniprocessor kernel whose threads are not preemptible. However, the problem can still arise in a kernel-based RPVM running on a multiprocessor computer system.

The external pager may also have outdated information concerning which pages are in main memory. This may result in the external pager making poor choices about the order in which to write pages back to disk. This is turn may result in more disk accesses than necessary. For example, consider the case where a set of pages {A, B, C, D} can be propagated in any order and where only page A is in the swap space. Since the external pager is not aware of page A's location, it may choose to propagate page A first. The paging in of page A may lead to page B being moved to the swap space. Next it chooses to propagate page B, resulting in page C being swapped out and so on. Instead of incurring only a single I/O operation to fetch page A from the swap space, the propagation of the set of four pages incurs a total of seven swap space I/O operations.

An external pager may need to buffer in virtual memory those database pages that cannot be propagated. As a result, it can also experience increased I/O costs associated with the double paging anomaly. This can result in a significant increase in the number of I/O operations incurred. Since avoiding this anomaly is
and B' (that is, A \textless B'), and the FB protocol (see Section 3.2) was adhered to, then B' would have been flush-locked while page A was being checked and propagated. However, page B was not flush-locked when a copy of it was acquired. Thus, propagating B' following further updates to A will not violate recoverable-persistence. Note that the order in which the copy of B is acquired and page A is propagated is important. If the copy of B is acquired after page A has been propagated, recoverable-persistence may not be preserved.

Similarly, additional rules and flush-locks that are added after the copy of B has been acquired need not be considered by the kernel thread propagating page B. For a large graph of dependent pages, a tree of such copies will have to be made but if copy-on-write is available the space and page copying overhead is only paid when the original pages are actually updated. Such a technique can be used to provide a variant of the \texttt{rpvm_propagate_request} call which allows concurrent database page updates, rule additions and page propagation to occur without compromising recoverable-persistence.

### 5.3 RPVM Data Structures

The data structures in the initial version of the RPVM module support a small set of operations on the pages of RPVM memory objects, namely:

- **User Interface Operations:**
  - Adding and removing P-FB rules;
  - Adding and removing flush-locks; and
  - Ordered propagation of a set of pages.

- **Kernel Operations:**
  - Determining the set of P-FB rules that defines the set of pages that potentially constrain a given page;
  - Determining whether a page is flush-locked;
  - Determining the pager from which to request a page
  - Recording the identity of modified pages that have been paged to the swap space; and
  - Ordered propagation of a set of pages.

For the initial prototype, we chose to implement relatively straightforward data structures, deferring tuning of the data structures until we had some performance numbers in hand. There are three main data structures associated with each RPVM memory object in this initial implementation:

- **SwapHash Table.** A table to record which pages belonging to the RPVM memory object have been diverted to the swap space. This is implemented as a simple hash table with chaining. An entry is inserted by the pageout daemon when it writes a dirty page to the swap space. This table is consulted by the page fault handler when a fault occurs for a page belonging to the RPVM memory object and when such pages that are not in main memory have to be propagated. An entry is removed when such a page is fetched from the swap space.

- **Flush-lock List and Tables.** An ordered linear list to record the ranges of pages that are explicitly prohibited from being propagated. To support fast flush-locking and removal of such locks, we have also implemented streamlined, shared memory-based flush-lock tables [5] which allow direct user access to flush-locks without kernel intervention. These structures are consulted by the page propagation procedure and pageout daemon before any page belonging to a RPVM memory object is propagated.

- **RuleTable.** A RuleTable contains the P-FB rule-based propagation constraints defined over the pages of the RPVM memory object. Conceptually, this is a two-dimensional array indexed by page offset within the object. It is in fact implemented by two hash tables. One hash table is indexed by offset, and its elements are linked lists of pages potentially constraining the page at the given offset (for example for page at offset B, the list of all pages A such that A \textless B). The other hash table is also indexed by offset, but its elements are linked lists of pages that can potentially be constrained by the page at the given offset (for example for page at offset A, the list of all pages...
page. Similarly, when a page fault occurs on a page in an RPVM memory object, the page fault
handler sends a message to the RPVM module requesting it to retrieve the page. Implemented this
way, the RPVM module is essentially an external pager resident within the kernel, except that it
may explicitly send pages to the swap space. For the purposes of our initial experiments, we have
taken this approach and the asynchronous messages are provisionally implemented as synchronous
procedure calls.

- **Bounded Checking.** In this approach the checking function is called synchronously. However,
to prevent the pageout daemon from blocking, a limit is placed on the number of propagation rules
that are checked. An upper bound on the number of pages accessed by the RPVM module function
can then be computed. As long as the sum of this upper bound and the original allowance made
for the pageout daemon is less than the size of the reserved memory pool [8] of the daemon,
blocking will not occur. If the threshold is reached, the function returns negatively, that is the
pageout daemon should handle the page as if it is constrained. If most pages are not constrained by
many rules, such an approach is practical.

### 5.2 Ordered Page Propagation.

Page replacement is only one case in which unconstrained RPVM memory object pages are sent to their
respective pages. The other cases are when a RPVM memory object is terminated, and when a user task
requests explicitly that a set of pages belonging to a RPVM memory object be propagated (by using the
`rpvm_propagate_request` call). In these cases the system must ensure that pages are propagated to the
database in the correct order, by constructing a graph of the dependencies between all concerned dirty pages.

The current implementation of the `rpvm_propagate_request` call assumes that all potentially
constraining pages that are required to be propagated are not updated from the time the call is made until it
returns. If not, recoverable-persistence may not be preserved. To understand why this is the case, consider
the implementation of the `rpvm_propagation_request` call. Let the dirty pages A and B are related by
the rule A ≤p B, and consider the actions taken when a user task requests that page B be propagated. Page
A is propagated before page B in order to maintain database recoverable-persistence. This is achieved by
making two `rpvm_memory_object_data_write` calls, one for page A followed by another for page B.

Now, consider the case where pages A and B are updated before the system call has been completed.
The updates may occur after the kernel thread yields the processor as a result of insufficient memory
resources while attempting to propagate page A using a `rpvm_memory_object_data_write` call. During the
time this thread is inactive, another user task may update pages A and B, and also require that the same
propagation constraint hold between these updates. Under the current implementation, when memory
resources become available, the waiting kernel thread completes the `rpvm_memory_object_data_write` of
page A, propagating the old version of page A. It then proceeds to propagate page B which contains new
updates. Thus, the new updates on page B are propagated before those of page A, resulting in a loss of
database recoverable-persistence. In a multiprocessor computer, this scenario can also occur even if the
kernel thread does not yield its processor.

A naive potential solution to this above problem, that does not involve disabling page access by user
tasks, is to have the kernel thread reexamine the state and propagatability of page A before proceeding with
the propagation of pages A and B. It will discover the new version of page A and may even discover that
additional pages may constrain either pages A or B. Such a reexamination of the page-flush policy and the
states of potentially constraining pages may be repeated indefinitely in the presence of continuous updates
and large graphs of dependent pages, leading to a situation where the actual propagation of page B never
takes place. Thus, such a technique is not desirable.

Another solution is for the kernel thread to acquire a copy of page B (if B is not flush-locked) before it
attempts to check the status of page A and propagate it if necessary. Thus, even if page B is updated again,
the version of page P that is eventually propagated by this thread will be the older and correct version, and
recoverable-persistence will have been maintained.

This technique allows page A to be updated after the kernel thread has determined it to be clean or has
propagated it, without compromising recoverable-persistence. This is because no propagation constraint
can exist between these later updates to page A and any update reflected in the copy of B (or B') that was
acquired prior to the processing of page A. If a P-FB constraint exists between the new version of page A
5.1 Changes To Mach Virtual Memory

Virtual memory in Mach is managed in terms of memory objects [7]. A memory object whose paging to and from its backing store is handled by an external pager is referred to as a permanent memory object. For instance, the files and databases that are memory-mapped by user tasks are permanent memory objects. In RPVM, we have extended such memory objects with an optional field to denote a special class of memory objects called RPVM memory objects. If the RPVM field is present, the pages belonging to the object are subject to the propagation constraints specified in the page-flush policy associated with it. A RPVM memory object may be shared by multiple tasks and its page-flush policy updated by any or all of them. However, coordination among the different tasks in user space with regards to updating the page-flush policy may be needed, for example to prevent deadlock or cyclically related page-flush rules [1]. As implementors of RPVM, we just have to be certain that user-level system calls provided by RPVM do not deadlock the kernel.

The interface between the existing virtual memory system and the RPVM module is straightforward. When a dirty page in a RPVM memory object is selected for replacement, the pageout daemon calls a function in the RPVM module to determine whether the page can be propagated safely (i.e., sent to the external pager). If not, that is it is constrained, the page is sent to the default memory manager to be written to the swap space. Doing so allows memory to be freed while maintaining recoverable persistence since the database is not updated and as far as the external pager is concerned, the page is still cached in main memory. Similarly, when a page fault occurs on a page in a RPVM memory object, the page fault code calls a function to determine whether the page should be retrieved from the external pager or the swap space. Memory management and the paging operations involving pages from all other regular permanent memory objects, and temporary memory objects (that are backed by the default memory manager) remain unchanged. In the rest of this section, we are concerned only with the pages of RPVM memory objects.

One key function of the RPVM module is to determine the propagatability of a database page that has been selected for replacement. If the page is constrained, it cannot be propagated. Since a user can specify arbitrary rules constraining the propagation of a page, and since the RPVM data structures reside in the kernel's virtual memory, the RPVM module can take an arbitrary length of time and memory to determine the propagatability of a page. If the pageout daemon synchronously calls this RPVM module function, in the worst case, all available free memory may be consumed before the call returns, causing the pageout daemon to block [8]. Since the pageout daemon is the only source of free pages, it cannot be blocked waiting for free pages. Hence, the daemon cannot check the propagatability of a page synchronously in this way. There are a few possible ways to address this propagatability checking problem:

- **Asynchronous Checking**. In this approach, the pageout daemon calls the function asynchronously - the pageout daemon sends an asynchronous message to the RPVM module, requesting it to determine the propagatability of a page selected for replacement. If the RPVM module is too slow to respond or physical memory is in short supply, the page may be paged to the swap space as if it was constrained, thus freeing the physical page. A variant of this technique that does not alter the replacement policy is for modified database pages in the inactive queue to be checked by a daemon before they are even selected for replacement. If such a page is determined to be propagatable, then it is flagged as so. If it is subsequently moved to the active queue and modified again, then the flag is reset. Whenever the pageout daemon selects a flagged and modified database page for replacement, it is propagated. Otherwise, the modified page is considered to be constrained and sent to the swap space.

- **Kernelized Pager**. In this approach, the pageout daemon also calls the function asynchronously but the RPVM module behaves much like an external pager. Rather than calling a boolean function directly to determine the propagatability status of the page, the pageout daemon sends an asynchronous message to the RPVM module, requesting it to dispose of the dirty page. If the RPVM module is too slow, or physical memory is in short supply, the page may become eligible for replacement again, and may be double-paged to the swap space, freeing the physical
tasks and the kernel [5]. To flush-lock a particular database page, a task simply turns on a bit in a flush-lock table that corresponds to its flush-lock. To flush-unlock, the bit is turned off. The kernel need only read the corresponding bit entry in the flush-lock table to determine the flush-lock status of any page. The kernel will never block when it does so, since setting a flush-lock bit can be performed in a single XOR operation. Such support can also be used to implement fast memoryful flush-locks in user space.

4.4 External Pager Interface

RPVM requires a simple extension to the external pager-kernel interface. In the existing Mach 3.0 external pager-kernel interface, the external pager provides a memory object data write procedure [19] that is commonly used to write a set of pages to their backing memory object on nonvolatile storage. The writes are neither guaranteed to be performed immediately nor in the order that they received by the external pager. For example, the UX server's pager attempts to optimize disk I/O by buffering and/or reordering file write requests. Thus, recoverable-persistence may not be maintained if RPVM makes use of such a procedure to propagate database pages. Moreover, the existing external pager-kernel interface does not allow the external pager to inform the kernel of successful propagations. Support for synchronous page propagation will be impossible.

Therefore, in addition to the kernel modifications described above, we also need to extend the external pager-kernel interface and modify external pagers to support the following:

- **Partially Ordered Page Propagation.** The external pager must guarantee that pages will be written to nonvolatile storage in either some fixed partial order or according to some partial order specified by the kernel. This will ensure recoverable-persistence as long as the partial order used does not violate page propagation constraints. In our prototype, we modified the inode pager of the UX server to provide a rpvms_memory_object_data_write procedure which guarantees that writes to nonvolatile storage are performed in the order in which write requests are received from the kernel. Although this ordering is overly strict, it allows recoverable persistence to be maintained. This procedure is similar to the memory_object_data_write procedure in all other respects.

- **Page Propagation Notification.** This allows the kernel or a user task to request that a notification be sent when a particular page propagation has been successfully performed. Such a facility enables the synchronous rpvms_propagation_request call to be supported by the kernel. In our prototype, the UX server's new rpvms_sync_memory_object_data_write procedure sends a message to a given port provided by the kernel when the write request has been performed successfully. It is similar to the rpvms_memory_object_data_write procedure in all other respects.

In the case of operating systems that do not have user-level external pagers, such support will have to be provided within their kernels and device drivers.

5. Implementation

We chose the Mach 3.0 kernel as the platform for our experiment. It can be argued that, rather than modifying the kernel, it would have been more appropriate and simpler to add RPVM support to an external pager (for instance, by augmenting the semantics of the Unix file system); however, we chose instead to make our changes directly in the kernel. We defer presenting the arguments in support of this choice until the next section, after we have described our implementation.

To implement RPVM in the kernel, two separate implementation issues had to be addressed. The first is the modifications to the virtual memory system to enable it to recognize the page-flush policies and to perform memory management and paging in a manner consistent with the policies. The second is the new kernel data structures that are needed to store and enable access to these page-flush policies. Essentially, a small RPVM module that provides data structures for storing page-flush policies and some functions for
4.2 Explicit Page Propagation

Tasks may need to explicitly request that a particular set of modified database pages be propagated, and moreover, may require that they be notified when such propagation requests have been successfully completed. For example, in many log-based recovery strategies, a transaction is considered to be committed if and only if certain log records have been propagated to stable storage. Thus, the RPVM interface also provides the following system call to support transaction semantics:

```c
kern_return_t rpvm_propagate_request(
    target_task, vm_task_t;
    begin_address, vm_address_t;
    size, vm_size_t;
    should_flush, boolean_t;
    should_rp_propagate, boolean_t;
    reply_port, mach_port_t);
```

The `rpvm_propagate_request` system call allows a task to request that all the pages in a range of pages that belong to a database and that have been modified be propagated. The range of pages is specified by the parameters `begin_address` and `size`. The pages in the range are also flushed from the kernel's cache if `should_flush` is set to TRUE. If `should_rp_propagate` is set to TRUE, propagation proceeds in an order that does not violate all relevant page-flush policies. Directly or indirectly constraining pages that are not in the range specified are also propagated. Due to propagation constraints that exist, some pages in the specified range may not be propagated. Such situations are indicated by a suitable return code.

This system call preserves recoverable-persistence only if the `should_rp_propagate` argument is TRUE and the pages that are propagated are not modified until the system call returns. A similar system call that does not require the second condition to hold in order to preserve recoverable-persistence may be useful and can be implemented but is not currently provided in our prototype.

The `rpvm_propagate_request` system call is an asynchronous call. The `reply_port` is provided to realize synchronous page propagation. A message from the external pager backing the database concerned is sent to the `reply_port` to signal the successful propagation of all propagatable pages in the specified range.

4.3 Other Related Extensions

In the previous sections, we described the basic system calls that make up the RPVM user interface. Extensions to this interface can allow propagation constraints to be specified and page-flush policies to be updated more concisely, but they do not increase the fundamental power of the model on which RPVM is based. Such extensions may be important in client-server systems where the above RPVM calls are made by remote client tasks.

For example, page-flush rules could be specified over sets of pages rather than over individual pages. Another extension would allow page-flush rules to be specified over pages belonging to different databases. In the current RPVM implementation, if propagation constraints exist between pages of different files on nonvolatile storage, the external pager has to handle and map the files in such a way that they appear to the kernel as a single database (or external memory object). This abstraction simplifies the RPVM and is consistent with our design goal of providing minimal but sufficient kernel support.

Another way to reduce the cost of processing page-flush rules and the cost of explicitly removing rules is to support automatic page-flush rule removal in the RPVM. Such a facility would allow tasks to terminate sooner than if they had to explicitly remove the rules that they have added. We will not describe the details of such a automatic rule removal scheme and its use in this paper. The `rpvm_add_rule` call would have to be extended to support such a facility.

Before a user task performs an update on a page, it must first flush-lock the page, unless the update can be performed atomically. After the update has been performed, the flush-lock can be removed. Thus, we can expect flush-locks on database pages to be taken very frequently and for extremely short intervals of time. Using the `rpvm_flush_lock` and `rpvm_flush_unlock` system calls will likely impose too high an overhead. A solution to this problem is to implement flush-lock tables that are shared between user
In our prototype RPVM system, we have extended the kernel interface to include the following new RPVM interface system calls:

```c
kern_return_t rpvms_flush_lock(
    target_task : vm_task_t;
    begin_address : vm_address_t;
    num_bytes : vm_size_t);
```

The `rpvms_flush_lock` system call flush-locks a range of pages that belong to a database.

```c
kern_return_t rpvms_flush_unlock(
    target_task : vm_task_t;
    begin_address : vm_address_t;
    num_bytes : vm_size_t);
```

The `rpvms_flush_unlock` system call removes the flush-locks of a range of pages that belong to a database.

In our prototype system, we implemented memoryless flush-locks. These flush-locks do not retain information about the identity of the tasks that added them or the number of such tasks. Such information may be required in systems where multiple tasks concurrently update and flush-lock shared database pages. Memoryless flush-locks would be inadequate in such situations. Non-memoryless or memoryful flush-locks could also be useful in handling and controlling errant task behavior. A memoryful flush-lock would keep a count of the number of times it has been added less the number of times it has been removed or unlocked. Such a flush-lock will be removed only if a `rpvms_flush_unlock` call results in this count being decremented to zero. However, since it is preferable to maintain the information associated with memoryful flush-locks in user-space, only memoryless flush-locks are supported directly.

```c
kern_return_t rpvms_add_rule(
    target_task : vm_task_t;
    rule : user_rule);
    /* struct[2] of vm_offset_t */
```

The `rpvms_add_rule` system call adds a P-FB rule to the page-flush policy of a database. The P-FB rule is specified in `rule`. The first element of `rule` identifies the potentially constraining page (i.e., page A of the P-FB rule, A ≤ p B). The second element of `rule` identifies the page whose propagation is potentially constrained (i.e. page B). The P-FB rule is added to the page-flush policy associated with the database to which page B belongs.

In our current implementation, both the potentially constraining and constrained pages of a P-FB rule are assumed to be from the same database. The identity of the `target_task` and the number of times a particular rule has been added are also not recorded. As in the case of memoryful flush-locks, such information may be needed in some environments and should be maintained in user-space.

```c
kern_return_t rpvms_remove_rule(
    target_task : vm_task_t;
    rule : user_rule);
```

The `rpvms_remove_rule` system call removes a P-FB rule from a page-flush policy of a database. The P-FB rule is specified in `rule`. The semantics of this system call are similar to those of `rpvms_add_rule` in all other aspects. The rule is removed from the policy of the database containing the potentially constrained page. It is the responsibility of the calling task to ascertain that removing the rule will not compromise recoverable-persistence.
program could cause the swap space to be filled with such pages, but we will not be concerned with this issue in this paper.

We refer to a virtual memory system that is extended in the above manner to support the RPU model as Recoverable-Persistent Virtual Memory (RPVM). The resultant RPVM maintains a page-flush policy for each database and guarantees that each database is recoverable-persistent by ensuring that database pages are paged out in a manner that does not violate any relevant propagation constraint. Page-flush policies are specified by user-level database and application systems.

The flush-locks taken and the P-FB page-flush rules specified depend on the specific recovery strategies being supported, as well as the granularity of updates and the order in which they are performed. The order in which these propagation constraints need to be added to a page-flush policy also depends on the relative ordering of the updates they constrain. We have developed a protocol, called the Flush-Before (FB) protocol [1], that specifies the flush-locks and page-flush rules that must be added to a page-flush policy and the order in which they are to be added relative to the updates under a variety of different conditions. For example, consider the situation in which an update to page A must be propagated either before or with the propagation of an update to page B, and page B's update is performed before page A's update. After page B's update is completed, page B must remain flush-locked until the page-flush rule A ≤ B has been added and page A has been flush-locked or updated. Page A's flush-lock can be removed as soon as its updates have been completed. If all database systems using RPVM adhere to this FB protocol, recoverable-persistence is ensured.

4. RPVM User Interface

Here we discuss how a database system or user-level task can make use of the facilities provided by RPVM to specify propagation constraints. Although this description of the user interface is tailored for use in the Mach environment, the interface can be easily supported by other operating systems provided they support virtual memory and file memory-mapping.

Before a database can be accessed using the VMDB approach, it must first be memory-mapped into the virtual memory address space of a task (for purposes of brevity, we will use the term task to include database systems). This is accomplished by the vm_map() call (or mmap() call in most UNIX operating systems). The page-flush policy for that database may be specified by that task or another related task (for example, the data manager task of a database system). In our prototype, a propagation constraint can only be added if the database pages to be constrained are currently memory-mapped into some task's virtual memory address space. The address location of a memory-mapped database page within a task's virtual address space is used to identify or name the database page.

More than one task can memory-map a database (external memory object semantics apply) and, likewise, more than one task may update the page-flush policy associated with the database. Unless explicitly removed, page-flush policies persist in the virtual memory system beyond the lifetimes of the tasks that update them, until all Mach kernel references to the database have been deallocated. No additional operations are required before a task can unmap a database or terminate. The existence of flush-locks beyond the lifetime of the tasks that took them may indicate an error condition but the handling of such errors is beyond the scope of this paper. The RPVM model does allow the virtual memory or some underlying system to provide atomic propagation and recovery of a selected set of database pages. This support can be activated whenever P-FB rules that form a cycle are encountered in a page-flush policy and all the pages involved are dirty. In our initial prototype, however, we are primarily concerned with handling P-FB rules that do not form cycles. Thus, neither atomic propagation of a set of pages nor recovery processing after a failure is implemented in the prototype.

4.1 Page-Flush Policy

Tasks can add and remove flush-locks and page-flush rules to and from page-flush policies. In our prototype, tasks do so explicitly via new system calls. While propagation constraints must be added in a timely fashion, they need not be removed. However, flush-locks and P-FB rules that are no longer necessary may unnecessarily constraint page propagation and add additional computational and I/O overhead to paging activities. Thus, such constraints should be removed.
into one by a recovery procedure. A database is recoverable-persistent if its state is always recoverable-persistent.

A key characteristic of recovery schemes for transaction management in database systems is the existence of ordering constraints on the propagation of updates to the database in nonvolatile storage. For example, in the Write-Ahead-Log protocol, the updates that modify data records must not be propagated before their corresponding log entries have been propagated. Propagation ordering constraints are also central to shadow paging, multi-level recovery [17] and other recovery schemes [13,14]. As a result of this observation, we conclude that in order for a database to be recoverable-persistent, its updates must be made persistent in some partial order. This partial order is application and recovery scheme specific, and in the RPU model, it is specified in the form of flush rules, which can be of two types:

- **Atomic Flush** (AF). Such a rule specifies that a set of updates be propagated atomically. We refer to such a set of updates as a Set of Atomic Persistence (SAP).
- **Flush Before** (FB). Such a rule specifies a propagation precedence relationship between two sets of updates. A FB rule is of the form $A \leq B$, where $A$ and $B$ are sets of updates, and it specifies that $A$ must either be propagated before $B$ is propagated or be propagated atomically with $B$. If a crash should occur and $B$ is persistent, then $A$ must also be persistent.

A flush policy is a set of flush rules and these flush rules are the central feature of our RPU model. It can be shown that these two types of flush rules are sufficiently powerful to express all propagation constraints, given our assumptions, and that the RPU model is sufficiently general to support the realization of a wide variety of recovery strategies [1].

### 3.2 Recoverable-Persistent Virtual Memory

In most virtual memory systems, the unit of data transfer between main memory and nonvolatile storage is the page. The virtual memory address space is made up of non-overlapping pages. Existing virtual memory systems neither support the notion of high-level updates nor guarantee the atomic propagation of updates that span multiple pages. Thus, if existing virtual memory is not to be drastically altered, direct support for flush rules over the propagation of updates in the RPU model is not possible.

Instead, we propose that the RPU model be supported in virtual memory by extending the kernel to support the following two types of page-level constraints on the propagation of virtual memory pages that are mapped to a database:

- **Flush-locks**. When taken, these prevent pages that are flush-locked from being propagated to the database. They are provided essentially to maintain intra-page consistency, since the virtual memory system is unable to detect when individual high-level updates begin and end in general. Unlike pinning, they are intended to be held for only the duration of individual high-level updates; and flush-locked pages may be paged to the swap space.

- **Page-Flush Before (P-FB) Rules**. Such a rule specifies a propagation precedence relationship between two pages. A P-FB rule is of the form $A \leq_p B$ where $A$ and $B$ are pages, and is used to specify that whenever pages $A$ and $B$ are dirty, page $A$ must either be propagated before $B$ is propagated or be atomically propagated with page $B$. Also, if page $A$ is flush-locked, page $B$ cannot be propagated. We can use P-FB rules to effectively specify the same propagation constraints as those that can be specified by the AF and FB flush rules of the RPU model. We refer to page $A$ as the potentially constraining page and page $B$ the potentially constrained page of the P-FB rule $A \leq_p B$. Page $A$ is a constraining page if it is a potentially constraining page and it either has been modified or is flush-locked. Page $B$ is a constrained page if it has been modified and page $A$ is a constraining page.

The set of flush-locks and P-FB rules pertaining to a database constitute the database’s page-flush policy. When a dirty page of a database is selected for replacement by the virtual memory system's replacement policy, but cannot be propagated according to the database's page-flush policy, it will be paged out to the swap space. Otherwise, it is written back to the database on nonvolatile storage. Conceivably, an errant
This approach incurs much messaging overhead. The TABS system implements both value and operation logging, together with the Write-Ahead Log protocol, to realize database recoverability.

Another example of a VMDB system is a database system built using Camelot [2], which takes advantage of Mach's external pagers [8] to provide recoverable virtual memory. In Mach, external pagers are responsible for writing the pages of memory-mapped files back to nonvolatile storage. In Camelot, the Disk Managers are responsible for propagating database pages in an order that does not violate propagation constraints. They are implemented as external pagers. However, since external pagers are user-level tasks with virtual memory, and pages that cannot be immediately propagated need to be buffered, Camelot's Disk Managers are essentially BPDB systems. Thus, Camelot can still suffer from double paging and the other problems that plague BPDB systems, albeit to a somewhat smaller degree.

Camelot also requires that a specific pin-modify-log protocol be followed by all database systems that modify the database. This protocol requires a number of messages to be exchanged between the database system and Camelot for every database update. Again, only value and operation logging is supported and the Camelot system restores the database to a consistent state after a failure before the database can be accessed. As a result of these characteristics, embedding Camelot in an operating system kernel will not yield a sufficiently general and flexible recoverable virtual memory system.

A commercial product that maps the database into virtual memory is ObjectStore [11]. However, ObjectStore does not use file memory-mapping. Instead, when a database page is accessed for the first time, a memory fault occurs and is serviced by ObjectStore, which requests the required page from a database server. Once a page has been cached in the database system's virtual memory address space, the cost of referencing data on that page is identical to that of a VMDB system. Also, modified pages can only be propagated back to the database server by ObjectStore. To the virtual memory system, database pages are no different from regular transient virtual memory. As a result, the buffer system architecture of a database system built using ObjectStore is very similar to that of a BPDB system with a buffer pool that is as large as the database. Hence, such a database system cannot fully realize all the benefits of the VMDB approach. Also, ObjectStore's database server uses a specific page-level log-based recovery strategy which is transparent to the database system.

In CPR [18], hardware support is provided for recoverable virtual memory. Unfortunately, the support is not based on a general recovery support model but implements a specific and limited recovery strategy. Recovery based only physical logging of entire database pages is supported.

Our work differs from the previous work in two ways: the recovery support we propose is flexible and general, and our prototype implements such support at the operating system kernel level, potentially realizing all the benefits of the VMDB approach.

3. The Model

Since the focus of this paper is on the implementation of kernel support for the RPU model in the Mach kernel, we will not attempt to justify that model in detail here [1]. Rather, we will only discuss the salient features of the model. We will also describe the additional semantics that are required for virtual memory to support this model.

3.1 Recoverable-Persistent Updates (RPU)

In the RPU model, all data accessed by a database system forms the database, including recovery-related data that is generated and maintained for the purposes of facilitating recovery after a crash or when the effects of updates need to be undone or redone. For example, log records generated by logging schemes are recovery data. Updates may affect any data in the database and since they are performed in volatile main memory, they must eventually be propagated to nonvolatile storage in order to be persistent. The model treats all data (and their updates) uniformly, whether they are recovery-related or not.

The database is memory-mapped into the virtual address space of the database system in the same fashion as memory-mapped files. The definition of consistency is application-specific and a database state is consistent as long as it is meaningful or useful to the application concerned. For example, a definition for consistency for most database systems is transaction-based atomicity and durability. We say that a persistent database state is recoverable-persistent if the state is either consistent, or it can be transformed

4
Our work explores the implications of a different approach to providing operating system support for recoverable-persistence in VMDB systems. Our approach is based on supporting a novel Recoverable-Persistent Updates (RPU) model [1] in the operating system kernel. The small set of simple primitives provided by the RPU model can be used to support a wide variety of recovery schemes, rather than only a specific strategy or class of strategies. The model allows greater flexibility with regards to the ordering of data record updates and recovery-related updates (e.g., logging). Thus, the RPU model is simple, general and flexible, and is highly suitable to being implemented in an operating system kernel. We refer to virtual memory that is extended to support the RPU model as Recoverable-Persistent Virtual Memory (RPVM).

VMDB systems built on top of RPVM can make use of both its file memory-mapping and recovery support facilities to realize all the benefits of the VMDB approach, without compromising their consistency requirements. Such VMDB systems can choose to implement their own recovery schemes using the primitives provided or to use those strategies provided by the RPVM system, if any is available. In the former case, the VMDB system is responsible for recovery processing following failures, while in the latter case, the underlying system uses some recovery scheme to restore the database to a consistent state on behalf of the VMDB system. In addition, such VMDB systems can specify that recoverable-persistence be maintained for only a selected part of the mapped database. For the remainder of the database, recoverable-persistence is not maintained and regular file memory-mapping semantics apply. The propagation constraints that are central to ensuring recoverable-persistence can be specified either explicitly via system calls or implicitly in the database. The relationship between the RPVM, database system and the rest of the computer system is depicted in Figure 1.

In this paper, we describe our initial experiences in modifying the virtual memory system of the Mach 3.0 kernel [19] to directly support this model. This involved adding a small RPVM module to the kernel, some small changes to the virtual memory page-in and page-out paths to communicate with this module, as well as the addition of a set of new system calls for manipulating the primitives provided by the RPVM module. From the point of view of the virtual memory system, the RPVM module looks very much like an external pager. This similarity will become clear as we describe the implementation. However, there are significant differences between the two and there exist compelling reasons for embedding the RPVM module in the kernel.

2. Existing Approaches

One approach, adopted by the TABS prototype [16], is to modify the virtual memory system to always consult a user-level recovery manager or process before it propagates any modified database page, say P. The recovery manager ensures that those pages that must be propagated before P have been propagated before sending a message to the kernel, informing it that P may be propagated. The page P is written back to the database only after the virtual memory system has received this message from the recovery manager.
1. Introduction

A key component of modern database systems is the buffer system, which attempts to keep the most frequently accessed parts of the database in a buffer pool maintained in main memory. Although most operating systems provide buffering facilities, these are typically insufficient to support the performance and consistency requirements of a database system. Hence, most existing production database systems provide their own buffer management. A Buffer Pool Database (BPDB) system allocates a buffer pool within its own virtual address space, and is responsible for its own buffer management. There are a variety of performance problems and other disadvantages associated with this approach [1,2]. For example, the paging policies of a BPDB system and the underlying virtual memory system may interact poorly, resulting in greatly increased I/O costs due to double paging [9]. The BPDB approach may also be unsuitable for use in object-oriented database systems and computing environments with either very large main memories or limited swap storage [1].

An attractive alternative is to extend the virtual memory system to allow database systems to use the buffering facilities of the operating system, without compromising the integrity of the database [6]. Database systems can exploit the buffering facilities of the underlying virtual memory system by mapping the database into virtual memory [2,10]. We refer to this approach as the Virtual Memory Database (VMDB) approach. Compared to a BPDB system, a VMDB system has a substantially simplified and smaller buffering component. More importantly, a VMDB system does not suffer from many of the problems associated with BPDB systems because the virtual memory manager has direct access to the database and other system resources like main memory, and has accurate utilization information about them. The database is not duplicated in the swap space and updates are more likely to be reflected in the database since the virtual memory system is able to page the database directly to and from the mass storage on which the database resides. Unexpected and high I/O overhead associated with double paging is eliminated without having to statically allocate main memory since a single page replacement policy is used. Referring a database object costs as little as referencing a transient target [11]. Virtual memory hardware can be exploited to speed up address translation and access control. Thus, the VMDB approach presents opportunities for improved performance, ease of programming and efficient use of resources in the memory hierarchy. Some examples of VMDB systems are TABS [16], Camelot [2], ObjectStore [11] and CPR [18].

A VMDB system, like a traditional database system, must be able to deal with failures and recovery. Updates modify database records in volatile virtual memory. System crashes can occur at any time, resulting in the loss of such data. As a result, the updates need to be made persistent on nonvolatile storage if they are to persist beyond the lifetimes of the programs that performed them. Also, updates may need to be undone or redone in order to restore the database to a consistent state [12]. If a database can be recovered to a consistent state in the event of such failures, then it is said to preserve the recoverable-persistence property and a database state that has this property is referred to as a recoverable-persistent state. Database systems use database recovery techniques that ensure that the database is always in a recoverable-persistent state and, if necessary, can transform such a state to an appropriately consistent state after recovery processing following system failure [13,14,15].

Essentially, recoverable-persistence is preserved by propagating updates to the database on nonvolatile secondary storage and imposing propagation ordering constraints on them. Since updates are propagated at the granularity level of pages, page propagation must be performed in such a way that these propagation constraints are maintained. Unfortunately, no such support for recovery and consistency exists in current day virtual memory systems. Page replacement is performed independent of these propagation constraints: for example, commonly encountered replacement policies seek to keep the most frequently used pages in memory. Hence, a VMDB system cannot simply be implemented on top of existing virtual memory systems. Existing VMDB systems solve this recovery problem by employing solutions that compromise some key benefits of the VMDB approach, and that support only either a particular recovery technique or class of recovery strategies. Thus, existing approaches are inadequate.
Kernel Support for Recoverable-Persistent Virtual Memory

Khien-Mien Chew
kenchew@cs.utexas.edu
Dept. of Computer Sciences, University of Texas, Austin, TX 78712-1188.

A. Jyothy Reddy
Hewlett-Packard Company, Cupertino, CA 95014.

Theodore H. Romer
Dept. of Computer Science and Engineering, University of Washington, Seattle, WA 98195.

Abraham Silberschatz
AT&T Bell Laboratories, 600 Mountain Ave., Murray Hill, NJ 07974.

February, 1993


Abstract

The buffering facilities typically provided by operating systems are not powerful enough to support the performance and consistency requirements of database systems. As a result, most database systems are structured as Buffer Pool Database (BPDB) systems, providing their own buffering facilities, with their own paging policies and recovery schemes. The emergence of operating systems with very large address spaces and flexible memory management makes Virtual Memory Database (VMDB) systems feasible. In such systems, the database is mapped into virtual memory and the buffering facilities of the underlying virtual memory system are used. VMDB systems do not experience many of the problems faced by BPDB systems. To support the consistency and recoverability requirements of VMDB systems, we have proposed that the virtual memory system be extended to support the Recoverable-Persistent Updates (RPU) model. This model is powerful and general enough to support a wide variety of policies for ensuring database recoverability. In this paper we discuss our approach to and progress in extending the Mach 3.0 kernel to provide direct support for this RPU model.

Keywords: virtual memory management, operating system recovery support, database system, Virtual Memory Database system, Recoverable-Persistent Virtual Memory, flush rules, flush-locks, propagation constraints, persistent storage, fault tolerance, Mach operating system, buffer management, memory-mapped database.

---

1 This material is based in part upon work supported by the Texas Advanced Technology Program under Grant No. ATP-024, the National Science Foundation under Grant No. IRI-9106450, and grants from the IBM and Hewlett-Packard Corporations.

2 Supported by a National Computer Board (Singapore) Postgraduate Scholarship.

3 Supported by a NSF Graduate Research Fellowship and an ARCS Fellowship

4 On leave from the University of Texas at Austin, where this work was performed.
KERNEL SUPPORT FOR
RECOVERABLE-PERSISTENT VIRTUAL MEMORY

Khien-Mien Chew, A. Jyothy Reddy,
Theodore H. Romer and Avi Silberschatz

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712-1188

TR-93-06 February 1993