Managing Discardable Pages with an External Pager

Indira Subramanian
Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Internet: indira@cs.cmu.edu

Abstract

We have designed and implemented an *external pager* that: (1) receives information regarding discardability of pages from the client such as a garbage collector for functional programming languages, (2) saves and restores only non-discardable pages and (3) influences page-replacement by pre-flushing discardable pages. In a general purpose operating system dirty pages are typically saved to disk and then restored from disk even when the application may not care about the contents of those pages. For example, copying garbage collection used in functional programming languages generates many pages that are dirty but discardable. Using the external pager to manage the discardable pages we observed that elapsed times for some applications decreased by as much as a factor of 6.

1. Introduction

Saving and restoring discardable dirty pages causes unnecessary disk traffic, thus preventing computer systems from achieving the best possible performance. An application may modify one or more pages during the course of its execution. At subsequent points of execution, the application may not care about the contents of some of these modified (dirty) pages. We refer to these modified pages as *discardable dirty* pages. The ratio between the average access time of magnetic disk and the average access time of main memory is often between 2,500 and 70,000 [Gibson 90]. Hence, saving and restoring discardable dirty pages would undermine the throughput of a computer system.

One of the applications that can take advantage of discardable page management is copying garbage collection. In systems that use copying garbage collection, such as the one described in [Appel 89], heap space is partioned into a *new*, *old* and *reserved* regions. Allocations are made out of the new space. When the new space is exhausted, a *minor* collection phase is initiated. During the minor collection, *live* data are copied from the new space to the reserved space, which immediately follows the old data. The old region continues to grow into the reserved region. Once the old region reaches a certain size, a *major* collection phase is initiated. During the major phase, the live data from the old region is copied to another region (in the heap), which becomes the old region from then on. At the end of each minor and major collections, several pages become discardable. Examples of systems that use copying garbage collection are: SML/NJ garbage collector [Appel and MacQueen 87], and the CMU Common LISP garbage collector [MacLachlan 91].

Another application domain that can benefit from support for discardable pages is image processing (e.g., image restoration, image compression, and texture synthesis). Texture synthesis is an important component of computer graphics as well as image compression. One of the techniques that has been developed for texture synthesis is a two stage procedure [Balram 91] involving matrix computations. This technique generates several intermediate matrices that become discardable. For example, consider the case

---

This research was sponsored in part by The Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title ‘‘Research on Parallel Computing’’, ARPA Order No. 7330, issued by DARPA/CMO under Contract MDS972-90-C-0035. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA, or the U.S. government.

where a typical image represented as a 512x512 double precision matrix is to be synthesised. At the end of the first stage, there will be 26 512x512 matrices of double precision data, of which 20 matrices are input to the second stage and 6 matrices become discardable. Hence, in the first stage, the total memory requirement is 52 megabytes of which 12 megabytes are discardable. In the second stage, there will be 21 matrices, of which 20 are from the first stage and 1 matrix is the output matrix of the second stage. As the iterations in the second stage proceed, each of the 20 input matrices become discardable. Support for managing discardable pages is important since images can be even larger. The ability to discard pages would also be useful in applications such as smoothing of noisy images, where a similar two stage procedure is used.

In Mach, the vm_allocate and vm_deallocate calls can be used to map or unmap pages in a task's address space [Rashid et. al. 88]. The vm_deallocate call could be used to unmap the pages that an application would like to discard. However, the physical pages get freed only if an entire object allocated by vm_allocate is being deallocated (by the vm_deallocate). If only a part of an object is deallocated, the pages are unmapped from the client's address space, but they remain as part of the object. Hence, vm_deallocate does not discard the pages unless a client deallocates an entire object. In fact, the copying garbage collector in CMU Common Lisp uses the vm_allocate call to allocate the new region; at the end of minor collection, this new region is deallocated by vm_deallocate call. Even modifying the kernel to solve this problem introduces additional difficulties because repeated vm_deallocates and vm_allocates can result in the fragmentation of a memory object, degrading memory management performance.

To the best of our knowledge, general purpose operating systems do not provide support for identifying discardable dirty pages as such and for avoiding the unnecessary saving and restoring of those pages.

2. Solution

Our solution is to exploit the knowledge about discardable pages in an application's address space and avoid unnecessary saving and restoring of those pages. Many applications, such as garbage collectors and image processing systems know what pages are discardable at any point in time. These applications can communicate this information to the operating system.

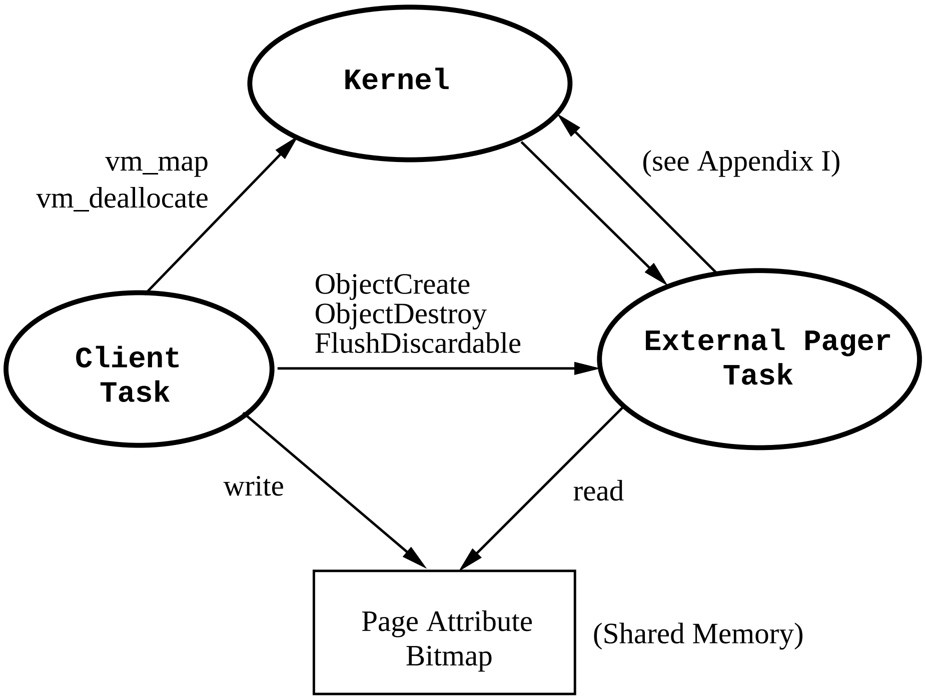
We have implemented under the Mach Operating System [Tevanian 87a], a pager that facilitates efficient management of discardable pages. Mach allows the user to create *memory objects* [Tevanian 87b] that can be managed by a user-level process, called the *external pager* [Young et. al. 87]. Through the external pager, a client may create an object and request that the kernel map the object into its address space. Figure 2-1 summarizes the interaction between the kernel, the pager and a client.

2.1. The Page Attribute Bitmap

The information regarding the discardability of pages in the memory object created by *ObjectCreate* is maintained in a bitmap shared between the client and the pager. This state information is written by the client and read by the pager; it contains one bit per page, indicating that the page is in one of two states:

- non-discardable - when paged out by the kernel, the pager must save the contents of this page to backing storage; when paging in, the pager must retrieve the contents of the page from the backing storage
- discardable - when paged out by the kernel, the pager may discard this page; when paging in, the contents of the page are irrelevant to the client

The page attribute bitmap, also called the *attribute object*, is itself a memory object created by the pager. The memory object whose pages will be managed by the pager, is called the *primary object*. A call to *ObjectCreate* returns two ports, one for the primary object and the other for the attribute object. The client then maps the primary object into its address space by calling *vm_map*, which results in the kernel calling *memory_object_init*. In *memory_object_init*, the pager maps the attribute object into pager's address space. Upon returning from *vm_map*, the client calls *vm_map* again to map the attribute object into client's address space. In this manner, the attribute object is shared between the client and the pager. The primary object and the attribute object are managed by two different threads running within the pager to avoid



<u>**Client to Kernel**</u>

vm_map	maps a memory object into client's address space
vm_deallocate	unmaps a specified range of pages from clients address space

<u>**Client to Pager**</u>

ObjectCreate	creates a memory object
ObjectDestroy	terminates a memory object
FlushDiscardable	flushes all discardable pages in a memory object

Figure 2-1: Interactions: Messages and Shared Memory

deadlock.

2.2. Improving Paging Performance: Pre-flushing Strategy

The pager achieves significant performance gains in two ways. First, by using the information in the shared bitmap, it can avoid saving and restoring discardable pages. Second, it can pre-flush discardable pages, allowing non-discardable pages to remain in memory for longer periods. This further improves performance because it is much less expensive to flush several discardable pages than to write a non-discardable page to backing store, and to read that page back in later. By default, the Mach kernel uses a global page replacement policy based on a modified First-In-First-Out strategy, which does not take into consideration whether a dirty page is discardable or not. We use two approaches for pre-flushing: *user-initiated* and *pager-initiated*.

In the user-initiated pre-flushing technique the client task, sets up the page attribute bitmap and explicitly requests the pager to flush discardable pages by calling FlushDiscardable. The pager then calls *memory_object_lock_request* to flush all the pages that are marked discardable.

In the pager-initiated pre-flushing technique, the pager calls *memory_object_lock_request* to flush a predefined number of discardable pages when it receives a *memory_object_data_write* call from the kernel. This is an appropriate time to flush discardable pages because if the kernel is making *memory_object_data_write* calls, physical memory must be tight. In the current implementation, each time, the pager makes flush requests to flush 64 pages. We chose 64 because pre-flushing too many discardable pages could add to the execution time; pre-flushing too few pages would mean that the kernel could pageout non-discardable pages. At this time, we have not experimented with values other than 64. The pager refrains from making flush requests if there are acknowledgements pending from previous *memory_object_lock_request* calls.

When the pre-flushing technique is being used, the client and the pager need to synchronize their access to the page attribute bitmap. A single lock is used for this purpose. The pager acquires the lock before making one or more flush requests to the kernel and releases the lock after it receives acknowledgements from the kernel for all of the requests. The client acquires the lock when it needs to change the state of one or more pages from discardable to non-discardable and then releases it. Note that when pre-flushing technique is not being used, access to the page attribute bitmap need not be synchronized.

3. Evaluation

Performance measurements of discardable page management by an external pager is important for answering three questions: First, are there significant performance gains to be realised even if only in a small number of important applications? Second, how many applications can benefit from support for discardable page management? Third, should the support for discardable page management be in the kernel, or in an external pager? The first and second questions will be answered as we build our experience with performance of applications such as the ones discussed in section 1. Even if the performance benefits from an in-kernel implementation are greater than from an external pager, we need to be convinced that (a) an in-kernel implementation is simple (b) there are enough applications to warrant an in-kernel implementation. In this section, we will present some data on the performance measurements we made in the context of a copying garbage collector.

3.1. Experimental Setup

The measurements that are reported here were made on a Digital Equipment Corporation's Decstation 5000/200 with 24 megabytes of physical memory, running Mach version 2.5. This is a typical configuration in our environment. For the purpose of running the tests, the machine was booted in single user mode. The only other programs running were a nameserver and the external pager. The nameserver is used only at client startup, to look up the external pager service port. Hence for all practical purposes, only the client and the external pager were running in the system.

We ran four configurations: (a) without pager, (b) with pager, not using pre-flushing, (c) with pager, using pager-initiated pre-flushing, and (d) with pager, using user-initiated pre-flushing. They are summarized in table 3-1. In each test case, each of the four configurations were run 6 times. The average measurements are being reported in the performance tables. The entries in the performance tables are described in table 3-2.

NoPager	run without the external pager
PagerNF	run with the pager, without pre-flushing
PagerPF	run with the pager, using pager-initiated pre-flushing only
PagerUF	run with the pager, using user-initiated pre-flushing only

Table 3-1: Configurations

3.2. Performance Measurements

Two test cases written in Standard ML [Milner et.al. 90] were studied: (a) compiling parts of the SML/NJ compiler and (b) a sort program. Modifications were made to the ML garbage collector to use our external pager to create and manage the heap memory.

Time	shown in minutes:seconds
Pageins:Read	number of pages read from disk
Pageins:NoRead	number of discarded pages returned to client as zero-filled
Pageouts:Write	number of non-discardable pages written to disk
Pageous:Discard	number of discardable pages written to pager
Pageouts:Flush	number of discardable pages requested of kernel to be flushed

Table 3-2: Performance Metrics

Besides exploiting support for discardable page management, the ML garbage collector does some memory management of its own. It takes a set of parameters called *gc* parameters, that help maintain the working set pertaining to the heap within the available physical memory [Cooper and Nettles 91]. The gc parameters have effect on how frequently minor and major collections happen. On a 24 megabyte machine, different combinations of gc parameters could result in variations in the elapsed time of a given application. Details regarding the impacts of gc parameters are beyond the scope of this paper. Cooper and Nettles report significant performance improvements running ML applications with various gc parameters when using our pager [Cooper and Nettles 91].

Performance measurements for the compiler are shown in table 3-3. We observe that:

1. The elapsed time when not using the pager is significantly larger than the user and system times (i.e., the CPU utilization is low). As we introduce the pager and then the pre-flushing techniques, the CPU utilization increases, due to reduced disk accesses.

2. When using the pager without pre-flushing, the number of reads and writes is higher than with user or pager initiated pre-flushing. This is because in both the pre-flush cases, the kernel is able to retain more non-discardable pages.

	Time			Pageins		Pageouts		
Configuration	User	System	Elapsed	Read	NoRead	Write	Discard	Flush
NoPager	2:49	0:16	16:00	22792	N/A	21693	N/A	N/A
PagerNF	2:51	0:17	8:01	6594	15516	6525	18813	N/A
PagerPF	2:45	0:18	3:56	710	24755	710	425	27899
PagerUF	2:49	0:22	5:18	2861	33051	2861	1720	36233

Table 3-3: Measurements: Compiling Parts of SML/NJ Compiler

The elapsed time for pager-initiated flushing is smaller than for user-initiated flushing. This is because the pager-initiated flush happens only when memory gets tight, and a small number of discardable pages are pre-flushed at that time. The user-initiated flush, on the other hand, happens whenever the user chooses to activate it, and the user may have marked many pages discardable. This implies that pager-initiated pre-flush is more likely to track closely the memory demands in the system than the user-initiated pre-flush. However, our experience has shown that this is not always the case. We do not in general, have an accurate knowledge of the optimal number of pages to pre-flush from outside the kernel. This is one of the disadvantages of doing discardable page management through an external pager. In the case of an in-kernel implementation, the kernel flushes a discardable page only when it is in need of more memory. Unlike the external pager implementation, there can never be too few or too many flushes.

Another test case was a sort program. It does quicksort using ML futures [Cooper and Morrisett 90], a variant of MultiLisp futures which is a construct for creating tasks and synchronizing among them [Halstead 85]. The results are summarized in table 3-4. Once again we find that using the pager, particularly with pre-flushing techniques, reduces the elapsed time significantly. Note that the elapsed times for pager-initiated flush and user-initiated flush are almost the same in this case.

Configuration	Time: User	Time: System	Time: Elapsed	Pageins: Read	Pageins: NoRead	Pageouts: Write	Pageouts: Discard	Pageouts: Flush
NoPager	1:36	0:16	24:04	27669	N/A	27524	N/A	N/A
PagerNF	1:36	0:19	8:42	10969	13439	12632	16038	N/A
PagerPF	1:37	0:16	4:00	3981	18463	5124	327	21454
PagerUF	1:36	0:18	3:58	3707	24491	5149	620	28327

Table 3-4: Measurements: Sorting using futures in ML

4. Related Work

Another example of an external pager is the PREMO pager [McNamee and Armstrong 90]. This work extends the external pager interface to implement user-level paging policies. The kernel is modified to supply the PREMO pager information on the state (such as modified, referenced etc.) of physical pages and accept from the pager the order in which the pages should be replaced.

5. Future Work

Managing discardable pages with an external pager has been shown to yield good results, especially when using pre-flushing techniques. However, it is evident that having the information about discardable pages within the kernel would have certain advantages:

- Performing too many pre-flushes is likely to impede system performance; in the case of in-kernel implementation, knowledge of physical memory availability would enable the kernel to flush a discardable page only when necessary, thereby avoiding unnecessary zero-fills
- It is possible to reduce the the number of zero-fills without compromising security. For example, in Mach, we could let the kernel keep track of which task had last used a discarded page in the inactive queue. When a task pagefaults on a discarded page, the kernel could return the discarded page in the inactive queue, which had previously belonged to the task thus avoiding the need to zero-fill the page. It is our belief that systems such as SML/NJ which page heavily against their own pages and generate many discardable pages will benefit from this approach.

To evaluate this approach, we have begun an in-kernel implementation of discardable page management under Mach version 3.0.

6. Conclusion

Given the wide disparity in the speed of primary memory and backing storage, it is important to manage primary memory efficiently. Physical memory is getting cheaper, but at the same time, we are seeing a growing number of applications that are memory intensive. In this paper we have demonstrated the need to manage discardable pages for a certain number of applications. We are in the process of identifying other applications that can benefit from support for discardable page management. Results from this effort and an in-kernel implementation will be reported in a future paper [Subramanian 91].

7. Acknowledgements

I would like to thank Scott Nettles for getting the SML/NJ working with the pager. I would like to thank Eric Cooper and Rick Rashid for suggesting the problem of discardable pages and for continued support in pursuing further work in this area. I am grateful to Jeannette Wing and Bob Baron for allowing me to use their workstations on which all the performance measurements were made.

Appendix I   Kernel - External Pager Interface

Our external pager is functional under Mach versions 2.5 as well as 3.0. At the time of writing, our pager uses only the Mach 2.5 Kernel - External Pager Interface.

kernel to pager:

memory_object_init
: initialise a memory object that is being mapped to a task's address space

memory_object_terminate
: indicates that the specified memory object is no longer mapped

memory_object_data_request
: request data from this memory object

memory_object_copy
: indicates that a copy has been made of the specified range of the given original memory object

memory_object_data_unlock
: request that the specified portion of the memory object be allowed the specified forms of access

memory_object_data_write
: write back modifications made to this portion of the memory object while in memory

memory_object_lock_completed
: indicate that a previous memory_object_lock_request has been completed

pager to kernel:

memory_object_set_attributes
: make decisions regarding the use of the specified memory object

memory_object_get_attributes
: retrieve attributes currently associated with an object

memory_object_data_provided
: provide data contents of a given memory object

memory_object_data_unavailable
: indicate that zero-fill page should be provided

memory_object_lock_request
: indicate that the protection on a given range of pages be changed; may require that the data be written back to the manager

memory_object_data_error
: indicate that a range of specified memory object cannot be provided at this time

memory_object_destroy
: indicate that the pager will no longer supply data for this object

References

[Appel 89] Andrew W. Appel.
Simple Generational Garbage Collection and Fast Allocation.
Software-Practice and Experience , February, 1989.

[Appel and MacQueen 87]
Andrew W. Appel and David B. MacQueen.
A Standard ML Compiler.
Functional Programming Languages and Computer Architecture.
Springer-Verlag, 1987, pages 301--324.
Volume 274 of Lecture Notes in Computer Science.

[Balram 91] Nikhil Balram and Jose M. F. Moura.
Recursive Enhancement of Noncausal Images.
In *Proceedings IEEE ICASSP'91*, pages 2997-3000. May, 1991.

[Cooper and Morrisett 90]
Eric C. Cooper and J. Gregory Morrisett.
Adding Threads to Standard ML.
Technical Report, School of Computer Science, Carnegie Mellon University, December, 1990.

[Cooper and Nettles 91]
Eric Cooper, Scott Nettles.
Experience With Mach External Pager for ML Garbage Collection.
In Preparation , 1991.

[Gibson 90] Garth A. Gibson.
Redundant Disk Arrays: Reliable, Parallel Secondary Storage.
PhD thesis, Computer Science Division, University of California, Berkeley, December, 1990.

[Halstead 85] Robert H. Halstead Jr.
Multilisp: A Language for Concurrent Symbolic Computation.
ACM Transactions on Programming Languages and Systems , October, 1985.

[MacLachlan 91] Robert A. MacLachlan.
CMU Common Lisp User's Manual
School of Computer Science, Carnegie Mellon University, 1991.

[McNamee and Armstrong 90]
Dylan McNamee and Katherine Armstrong.
Extending the Mach External Pager Interface to Accommodate User-Level Page Replacement Policies.
In *USENIX Mach Workshop*. USENIX Association, October, 1990.

[Milner et.al. 90] Robin Milner, Mads Tofte, and Robert Harper.
The Definition of Standard ML.
MIT Press, 1990.

[Rashid et. al. 88] Rashid, Richard F. et. al.
Machine Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures.
IEEE Transactions on Computers , August, 1988.

[Subramanian 91] Indira Subramanian.
Managing Discardable Pages.
In Preparation , 1991.

[Tevanian 87a] Avadis Tevanian Jr. and Richard F. Rashid.
MACH: A Basis for Future UNIX Development.
Technical Report, Computer Science Department, Carnegie Mellon University, June, 1987.

[Tevanian 87b] Avadis Tevanian Jr.
Architecture-Independent Virtual Memory Management for Parallel and Distributed Environments: The Mach Approach.
PhD thesis, Computer Science Department, Carnegie Mellon University, December, 1987.

[Young et. al. 87] Michael Young et. al.
The Duality of Memory and Communication in the Implementation of a Multiprocessor Oprating System.
Technical Report, Department of Computer Science, Carnegie Mellon University, August, 1987.

Table of Contents

List of Figures

List of Tables