Reducing the Virtual Memory Overhead of Swizzling

Vivek Narasayya, Tze Sing Eugene Ng, Dylan McNamee, Ashutosh Tiwary, Hank Levy
University of Washington, Seattle.
{nara,eugeneng,dylan,tiwary,levy}@cs.washington.edu

Abstract

Swizzling is a mechanism used by OODBs and persistent object systems to convert pointers from their disk format to a more efficient in-memory format. Previous studies of swizzling have focussed on analyzing the CPU overhead of pointer translation and studying trade-offs in different approaches to swizzling. In this paper, we show that there is an additional indirect but important cost associated with swizzling: swizzling a read-only page causes it to be "dirty" with respect to the operating system. At the onset of paging, these read-only pages may be written to the swap file unnecessarily. We propose a simple modification to the operating system that reduces the impact of this overhead on application performance.

1 Introduction

In recent years object oriented databases (OODBs) have emerged to meet the demands of applications such as computer-aided design and manufacturing (CAD/CAM), computer-aided software engineering (CASE), multimedia, and network management. Most OODBs use swizzling to improve performance when accessing persistent objects in memory. Swizzling converts pointers from their disk format (an object identifier) to a more efficient in-memory format (a virtual memory address) when persistent objects are brought into memory by the OODB.

Swizzling is a complex technique with a number of dimensions [7]. Previous swizzling studies have analyzed the costs of pointer translation [5] and explored trade-offs such as software vs. virtual memory hardware swizzling, eager vs. lazy swizzling, and object grain vs. page grain swizzling [8]. These studies do not identify or quantify an important indirect cost of swizzling that occurs due to the interaction of swizzled pages with the operating system (OS). Since swizzling modifies pointers on a page, the OS considers a swizzled page to be dirty, even though the application may consider it read-only. When memory pressure on the machine causes paging, these dirty pages can be paged out to the swap file unnecessarily[1]. For applications that access large data sets, this paging cost can dominate the other costs of swizzling and affect application performance. We call these unnecessary page-outs the *virtual memory overhead* (VM overhead) of swizzling, since they are caused by the interaction of swizzling and the virtual memory system.

In the next section we describe and quantify the VM overhead problem in the context of Texas [6], an OODB that employs pointer swizzling using virtual memory hardware. In Section 3 we propose a solution to this problem that involves a small change to the operating system, and argue that it can be used to improve performance of predominantly read-only workloads. In Section 4 we discuss how our technique can be used to solve other problems of the same general nature. We summarize and conclude in Section 5.

2 Quantifying the VM overhead

VM overhead can occur in any OODB that swizzles. In this section we describe and quantify VM overhead in Texas for the OO7 workload.

2.1 Swizzling and Buffer Management in Texas

Texas uses pointer swizzling at page fault time [10] to provide a mapping from persistent to virtual memory. To ensure that the first access to a page is intercepted, Texas access protects the page. When a page fault occurs, Texas swizzles the page by overwriting pointers in the persistent format with virtual addresses. Since swizzling overwrites pointers on a page, the page becomes dirty with respect to the VM system, even though the application never directly modifies the page. Texas does not explicitly control the caching of pages from the database. It reads database pages into the file system buffer, copies it from there into the application's memory, and swizzles it. These dirty pages

[1] Note that many OODBs try to control paging of application data by managing their own buffer pool. However, even such systems can perform unnecessary paging if the OS pages-out dirty pages in the buffer pool. This is described in Section 4.

are backed by the swap file and managed by the operating system.

We identified three main costs of swizzling in Texas:

- *Signal handling cost*. This is the cost of delivering a signal to Texas' signal handler. This shows up as CPU time of the application.
- *Pointer translation cost*. This is the cost of translating pointers in a page. It includes the costs of locating objects on the page, getting type information for each object, potentially reserving virtual address space for pages pointed to by pointers in this page, recording mapping between persistent and virtual pages, and finally coverting the object identifier into a virtual memory pointer. Once again this appears as CPU time of the application.
- *VM overhead*. This happens when a read-only page of the application is faulted in[2] and swizzled by Texas, which makes the page dirty. This *read-only* page may get written to the swap file due to paging. *A key observation is that this read-only page could have been discarded instead of being paged out; i.e. if the application touched the page again, it is read in from the database and re-swizzled.* These extra page-outs of read-only pages is the VM overhead. Re-swizzling the page results in better performance, because the cost of swizzling a page is much less than the cost of a page-out.

2.2 Experiments and results

We ran an experiment to determine the impact of VM overhead in Texas for a read-only workload. The hardware used is a DECStation 3000/400 running Digital Unix v.3.2 with 64 MB of memory; the database is stored on a DEC RZ56C disk and the swap file is on a DEC RZ26C disk. The numbers reported are the averages of three separate runs. We used the T1 traversal of the OO7 benchmark [2] as our read-only workload and varied the database size from 23 MB to 86 MB.

A read-only application should never have to page-out data. Therefore, the page-outs measured in the experiment are, by our definition, the VM overhead in Texas. Table 1 shows the number of page-outs as the number of data pages accessed increases from 2759 to 7551. For the first four database sizes, there are no page-outs because the data fits into memory[3]. For larger database sizes, the ratio of the number of page-outs to the number of pages accessed by the application increases steadily to about 30%. This means that for certain database sizes, almost a third of all pages accessed from the database are written to the swap file during the execution of the program. The actual number of page-outs for a given program is determined by a combination of the number of pages accessed, locality in the access pattern, the amount of physical memory available, and the page replacement policy. In the T1 traversal, there is a good deal of locality in the access pattern. Therefore, in a workload with less locality (e.g. sparse, sequential access), we expect the ratio of page-outs to pages accessed to be even higher than 30%.

Table 1: OO7 T1 Traversal. Number of page-outs

Database size (MB)	23	30	37	44	51	58	65	72	79	86
Pages Accessed	2759	3109	3722	4334	4878	5411	5957	6485	7021	7551
Page-outs	0	0	0	0	258	707	1134	1596	1902	2230

To summarize, a read-only application should *never* have to page-out data. However, our experiments have shown that the interaction of swizzling and the VM system in Texas can cause a substantial number of page-outs.

3 Proposed Solutions

3.1 Clearing the Dirty Status of a Page

The VM overhead arises because the OS does not know that the application wants particular pages to be treated as clean even though they were modified. We therefore propose a simple system call that allows the application to request the OS to clear the dirty status of a page. If a clean page is replaced from memory, it is discarded and never written to the swap file. If the application references such a page, it needs to re-swizzle that page. In Texas, this is can be achieved by having the system call also reset the protection bit for the page. If the page is referenced again, a fault occurs and the signal handler re-swizzles the page. In the general case, the application requires a notification when a cleaned and discarded page is referenced. This notificaton can either be a signal or an upcall into the application.

We intend to implement this system call in Digital Unix. We will have to handle two types of pages: anonymous memory pages which are managed by the VM system, and mapped file pages which are handled by the Unified Buffer Cache (UBC). In both cases we will need to reset the dirty

[2] The page is copied from the file system buffer cache into the application's memory. This copy causes the page to be dirty; and so does swizzling. However, even if Texas used a different approach to reading data (e.g. a mapped file), swizzling would still cause the page to be dirty.

[3] Note that the operating system takes up between 3000 to 4000 memory pages.

bit in the process' virtual memory map (**vm_map**) and the corresponding bit in the physical map (**pmap**). However, in the case of a mapped file, the page must also be moved from the UBC's dirty list to the clean list. We expect the cost of executing the system call itself to be very small (a few usecs).

The exact impact of the VM overhead on the application depends on how much of the page-out time can be masked by overlapping it with computation in the program. Since this is hard to quantify precisely, we estimated the average time for a page-out using a separate microbenchmark. This program allocated a large number of pages and modified one byte in each page. It did no other computation. We also estimated the cost of swizzling a page in Texas. The results are shown in Table 2. The time saved by the application using our system call depends on whether or not a page that has been cleaned and replaced, is referenced again by the application. If the page *is* referenced again, then the application saves 10 ms on the average; if not it saves the full 13 ms.

Table 2: Cost of swizzling vs Cost of page-out

Average swizzling cost per page in Texas	Estimated time for a page-out using a micro-benchmark
3 ms	13 ms

3.2 Alternatives

An alternative to clearing the dirty bit is to avoid swizzling in the first place. OODBs, such as QuickStore [9] and ObjectStore [3], attempt to avoid swizzling by locating the database segment at the same virtual address of the application each time. However, avoiding swizzling alone doesn't avoid VM overhead, because pages are still dirty due to copying from the file system buffer. Integrating buffer management with virtual memory [4] *and* avoiding swizzling would eliminate VM overhead for these systems.

Another way to avoid VM overhead is to map a file backed by an NFS server that provides clean swizzled pages to the application. The drawback of this scheme is that it requires extensive modifications to the NFS server. Alternatively, an OS like Mach allows the use of external pagers, which can be used to swizzle and page and present it to the application in a clean state. The drawback of this approach is the non-portability of such OS facilities.

4 Other Applications of Our Technique

Our dirty bit clearing technique can be used in conjunction with different buffer management strategies. For example, QuickStore manages a fixed size buffer pool in virtual memory and assumes that the buffer is always backed by physical memory. (This is similar to the technique used by relational databases). As long as this assumption is not violated, there is no VM overhead because the database system is responsible for paging in the buffer pool, and it knows which pages are read-only. However, this assumption can be violated in the presence of memory competition, and read-only pages in the buffer pool that are dirty due to swizzling may get paged out. Making the pages in the buffer pool clean prevents paging of read-only data.

OODB buffer management can be integrated with virtual memory [1] [4] by mapping the database into the virtual memory of the application. The virtual memory system is responsible for paging data to and from the mapped file. Once again, swizzling causes read-only pages to be dirty. For correctness, these swizzled pages cannot be allowed to get paged back to the database. Hence, they have to be treated by the OODB as read-write pages, which results in higher overhead. Once again, our technique can be used to avoid this overhead.

The ability to clear the dirty status of a page can be used for other problems of the same general nature as the VM overhead problem. In general, whenever it is cheaper to "recompute a page" as opposed to paging it out, our technique is applicable. An example of this is a graphics application where data on a page is decompressed for the purposes of display.

5 Conclusion

In this paper we have identified an indirect cost of swizzling called VM overhead, which arises when read-only pages are swizzled and become dirty with respect to the OS. We have quantified the VM overhead in the context of Texas. We propose a simple system call to clear the dirty status of a page, that can be used to reduce the VM overhead. We intend to implement this system call in Digital Unix.

References

[1] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Vlduriez. Prototyping bubba, a highly parallel database system. *IEEE Transactions on Knowledge and Data Engineering*, 1(2):4–24, 1990.

[2] M. J. Carey, D. J. Dewitt, and J. F. Naughton. The oo7 benchmark. *1993 ACM Sigmod. International Conference on Management of Data*, 2(22):12–21, May 1993.

[3] C. Lamb, G. Landis, J. Orenstein, and D. Weinred. The objectstore database system. *Communications of the ACM*, 10(34), October 1991.

[4] D. McNamee, V. Narasayya, A. Tiwary, H. Levy, J. Chase, and Y. Gao. Virtual memory alternatives for client buffer management in transaction system. *Submitted for Publication*, 1996.

[5] J. E. B. Moss. Working with persistent objects: To swizzle or not to swizzle. *IEEE Transactions on Software Engineering*, 8(18):657–673, August 1992.

[6] V. Singhal, S. V. Kakkad, and P. R. Wilson. Texas: An efficient, portable persistent store. *Proceedings of Fifth International Workshop on Persistent Object Sytems*, September 1992.

[7] S. J. White. Pointer swizzling techniques for object-oriented database systems. *Ph.d. Thesis. University of Wisconsin, Madison*, 1994.

[8] S. J. White and D. J. Dewitt. A performance study of alternative object faulting and pointer swizzling strategies. *Proceedings of the 18th VLDB Conference, Vancouver, Canada*, August 1992.

[9] S. J. White and D. J. Dewitt. Quickstore: A high performance mapped object store. *Proceedings of the 1994 ACM-SIGMOD Conference on the Management of Data , Minneapolis, MN*, May 1994.

[10] P. R. Wilson. Pointer swizzling at page fault time: Efficiently supporting huge address spaces on standard hardware. *ACM SIGARCH Computer Architecture News*, 4(19), June 1991.