Page Replacement and Reference Bit Emulation in Mach

Richard P. Draves

rpd@cs.cmu.edu
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue Pittsburgh, PA 15213

Abstract

A page-replacement algorithm guides the operating system when it needs to create more free pages by reclaiming pages that are filled with data. The Mach kernel approximates the familiar Least Recently Used (LRU) algorithm with an algorithm known as FIFO with Second Chance. This strategy uses page-referenced information to avoid reclaiming recently-referenced pages. On hardware lacking such support, the kernel must detect references in software.

This paper describes the Mach kernel's page-replacement algorithm and considers three software techniques for detecting references. It shows that Mach 3.0's *reference fault* technique outperforms Mach 2.5's *reactivation fault* technique, and also outperforms reference detection via a software TLB miss handler. Based on the performance of the Mach 3.0 implementation, we conclude that hardware page-referenced information is not a prerequisite for a satisfactory page-replacement algorithm.

1 Introduction

The Mach kernel uses a simple global page-replacement algorithm, called FIFO with Second Chance, that requires page-referenced information to approximate LRU behavior. The Mach 2.5 implementation performed unsatisfactorily on architectures without hardware page-referenced support. The Mach 3.0 kernel uses machine-independent reference bits and *reference faults* to emulate page-referenced support. This implementation provides an adequate LRU approximation with low overhead, leading to the conclusion that hardware page-referenced information is not a requirement for a satisfactory page-replacement algorithm.

Page replacement is a very old problem [Belady 66, Denning 68] that has received little attention in recent years. This is for good reason. The choice of page-replacement algorithm largely doesn't matter, because memory is cheap and most machines have enough memory that paging is not a concern. If a machine clearly doesn't have enough memory to support its work-load, then no page-replacement algorithm can rescue performance. However, a page-replacement algorithm should still meet several criteria:

This research was supported in part by a fellowship from the Fannie and John Hertz Foundation and 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.

- *Low Overhead* — Because in many environments very little page-replacement occurs, the algorithm should not impose a noticeable overhead when it is not needed.
- *No Tuning* — An algorithm that requires a human other than the original designer to set parameters will perform poorly in the real world. Machine-dependent and user-dependent tuning is undesirable.
- *LRU Approximation* — When it is needed, the page-replacement algorithm should avoid pathological behavior. In a slightly memory-poor environment, an LRU approximation that keeps the system's working set resident will make a difference.

For example, as described later in Section 2.3, tuning Mach's page-replacement algorithm to produce a better LRU approximation improved the performance of a simple compilation test by a factor of four.

This paper describes the Mach kernel's page-replacement algorithm. In Section 2 we give an overview of the algorithm's operation and point out some problems with the emulation of hardware page-referenced information. We examine TLB-based emulation in Section 3 and then present Mach 3.0's machine-independent emulation in Section 4. In Section 5, we discuss the influence of Mach's external memory management interface on the page-replacement algorithm. We consider related work in Section 6. Finally, in Section 7 we summarize the paper.

2 Page Replacement in Mach

The Mach kernel uses a FIFO with Second Chance page-replacement algorithm. This algorithm approximates the performance of LRU page-replacement with the efficiency of FIFO page-replacement. The machine-independent pageout daemon, which implements the algorithm, uses Mach's pmap interface to manipulate machine-dependent page-referenced information. On architectures without such information, the pmap module and the pageout daemon must interact to emulate page-referenced support.

Mach's page-replacement algorithm has two major advantages:

- *Simplicity* — The algorithm attempts to reclaim a reasonable page, not the optimal page. A small amount of code and runtime effort produces good performance.
- *Scalability* — The performance of the algorithm is relatively insensitive to the actual amount of physical memory, because the pageout daemon never scans physical memory directly. Instead, it operates on queues of physical pages.

Unfortunately, the Mach 2.5 implementation of the algorithm proved to be difficult to tune properly on architectures without page-referenced support, and consequently it performed unsatisfactorily on those architectures.

2.1 FIFO with Second Chance

The Mach implementation of FIFO with Second Chance uses three queues of physical pages, known as the free queue, the active queue, and the inactive queue. The pageout daemon, a system thread running inside the kernel's address space, moves pages from the inactive queue to the free queue and from the active queue to the inactive queue.

To manage physical memory, the kernel manipulates small page structures. Each page structure represents one page of physical memory. The structure records various attributes of the physical page, including its address, and contains link fields for the paging queues.

Pages on the free queue contain no useful data. All pages on the free queue are equivalent. The kernel allocates pages from the free queue to satisfy page faults or to perform internal data structure allocation. Zero-filling, if necessary, happens after a page is taken from the free queue. If there are insufficient free pages, an allocating thread waits for the pageout daemon to create more free pages.

Pages on the active queue are mapped by some process and are assumed to belong to the system's working set. The active queue is FIFO; new pages start at the back of the queue and the pageout daemon takes pages from the front of the active queue.

The inactive queue, which is also FIFO, acts as a buffer between the free queue and the active queue. Inactive pages are assumed to be less valuable than active pages. Pages which are referenced while on the inactive queue return to the active queue instead of moving to the free queue. The inactive queue implements the page-replacement algorithm's LRU approximation by preventing frequently-referenced pages from being reclaimed.

The pageout daemon's primary responsibility is creating free pages. It wakes up when the free queue falls below `vm_page_free_min` pages, and runs until the free queue contains at least `vm_page_free_target` pages. The kernel uses these thresholds to reduce the overhead of repeatedly invoking the pageout daemon. Once awake, the pageout daemon scans the inactive queue. Clean pages are immediately moved to the free queue, and dirty pages are first cleaned by writing them to backing store. After the free queue reaches `vm_page_free_target` (or if the inactive queue is empty), the pageout daemon refills the inactive queue from the active queue, until it reaches `vm_page_inactive_target`.

Figure 1 shows the movement of pages among the active, inactive, and free queues.

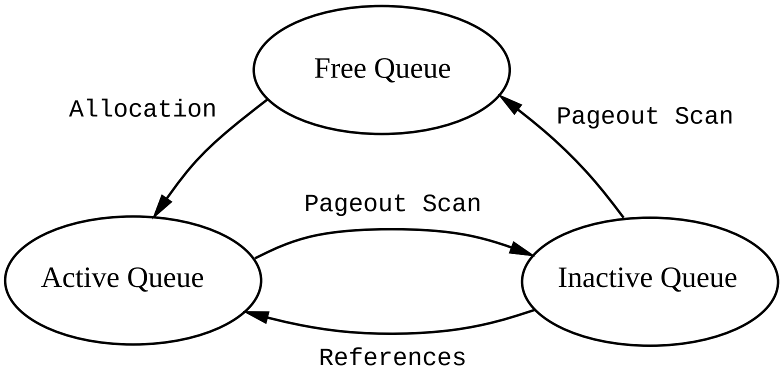

Figure 1: The Paging Queues

2.2 Reference Bits

The FIFO with Second Chance algorithm relies on being able to detect references to pages on the inactive queue. The Mach kernel makes use of page-referenced information, if it is provided by the hardware, through two pmap functions. If the hardware does not directly supply page-referenced information then the kernel must somehow emulate it. As done in Mach 2.5, this leads to two very different modes of operation for the page-replacement algorithm.

The Mach kernel accesses MMU hardware through a machine-independent interface, the pmap interface [Rashid et al. 87, Tevanian 87]. A pmap is the machine-dependent representation of an address space. The pmap interface contains, for example, functions to create and destroy pmaps and enter virtual-to-physical mappings into a pmap. The kernel's pmap module implements the pmap interface.

Some hardware architectures provide page-referenced information. On architectures with page tables, this commonly takes the form of a reference bit in each page-table entry. The operating system can clear the bit in a page-table entry and virtual memory references through a page-table entry set the bit. However, other forms of page-referenced information are possible. For example, the inverted page table found in the IBM RT [IBM 88] has a single reference bit per physical page.

The pmap module must provide two functions on a physical page address, to clear and query reference information, as shown in Figure 2. Because the interface operates on physical pages instead of virtual pages, the implementation on architectures with reference bits in page-table entries must examine or modify the reference bit in every page-table entry mapping the specified physical page. On architectures without hardware page-referenced information, `pmap_clear_reference` should remove all mappings for the physical page and `pmap_is_referenced` should always return FALSE.

`void pmap_clear_reference(physical address)`
: Clear machine-dependent page-referenced information associated with the specified physical page.

`boolean_t pmap_is_referenced(physical address)`
: Query machine-dependent page-referenced information associated with the specified physical page, returning TRUE if the physical page has been accessed since the page-referenced information was last cleared.

Figure 2: Page-Referenced Functions

Using these pmap functions, the Mach 2.5 pageout daemon functions as shown in Figure 3. The page-replacement algorithm behaves quite differently on machines with and without hardware page-referenced support:

- On machines without such support, pages on the inactive queue are not mapped into any address space. Faults on pages in the inactive queue, known as *reactivation faults*, are satisfied without performing I/O, by moving the page back to the active queue and remapping it.

- On machines with such support, pages on the inactive queue are left mapped into address spaces. Instead of using reactivation faults to move pages back to the active queue, the pageout daemon notices before freeing a page that it has been referenced, and activates the page instead. This type of reactivation may be viewed as the lazy evaluation of reactivation faults.

2.3 A Problem

Mach's first platform was the VAX-11/784, a four-processor version of the VAX-11/780. The VAX architecture [DEC 79] does not provide hardware reference bits. As originally conceived [Tevanian 87, pages 43–45], the inactive queue contained pages not mapped into any

```
vm_pageout_scan() {
        while (vm_page_free_count < vm_page_free_target) {
                page = dequeue(vm_page_inactive_queue);
                if (pmap_is_referenced(page->phys_addr)) {
                        enqueue(vm_page_active_queue, page);
                } else {
                        clean the page if it is dirty;
                        enqueue(vm_page_free_queue, page);
                }
        }

        while (vm_page_inactive_count < vm_page_inactive_target) {
                page = dequeue(vm_page_active_queue);
                pmap_clear_reference(page->phys_addr);
                enqueue(vm_page_inactive_queue);
        }
}
```

Figure 3: The Mach 2.5 Pageout Daemon

address space, and at boot-time the implementation configured `vm_page_inactive_target` to a small value, never larger than 80 pages. Once configured, the implementation never changed `vm_page_inactive_target`.

After some experience with the page-replacement algorithm, it became clear that the original configuration of the `vm_page_inactive_target` was not appropriate for machines with hardware page-referenced support. In fact, it was desirable to make the inactive queue larger than the active queue. A large inactive queue improves the quality of the LRU approximation by giving pages more time in which to be referenced. When the paging rate is high and the pageout daemon runs frequently, a small inactive queue is ineffective because relatively few pages are reactivated. With a small inactive queue and a high paging rate, the page-replacement algorithm degenerates to almost pure FIFO replacement. However, a reasonable LRU approximation is most important when the paging rate is high, because it is only by finding the system's working set and keeping it resident that the paging rate will come down.

To examine this effect, we used a small compilation test suite on an DECstation 3100 (DS3100) configured to use 8 megabytes of memory. The DS3100 has a 16.67Mhz MIPS-architecture R2000 CPU. The MIPS architecture does not provide page-referenced information. The compilation test compiles nine small C programs.

We ran the compilation test nine consecutive times while varying the inactive target. Figure 4 summarizes the results. With a small inactive target of 20 pages out of 1100 available pages, the Mach 2.5 algorithm performed poorly, leading to excessive paging. With an inactive target of 700 pages, the test completed approximately four times faster. Even after the inactive target was lowered to the original small value, the test completed more quickly because the smaller inactive queue became more effective once the paging rate had dropped. For comparison, the test took 19.2s to complete when run single-user on a DS3100 with 16 megabytes of memory,

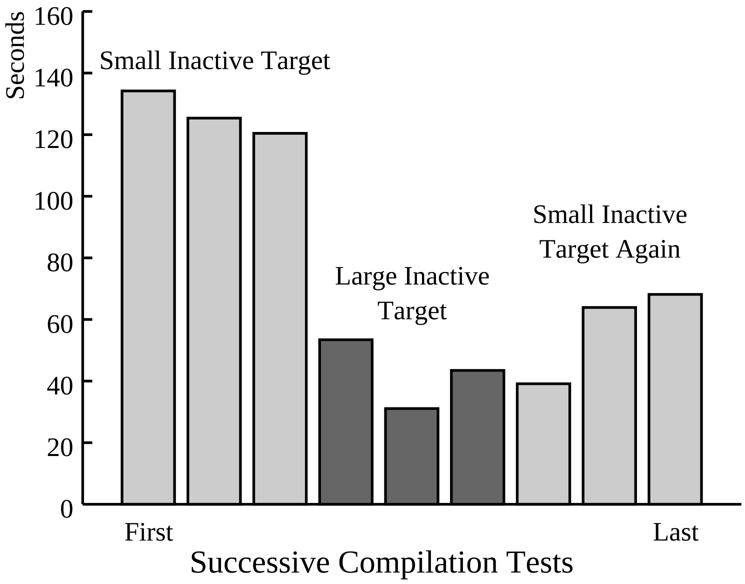

Figure 4: A Small Inactive Target Leads To Thrashing

Unfortunately, a large inactive queue is not a panacea. On machines without hardware reference bits, pages on the inactive queue are unmapped. A large inactive queue on these machines reduces the amount of physical memory directly available to applications, leading to *reactivation fault thrashing*. This can occur when an application's working set resides in physical memory but exceeds the capacity of the active queue. On the DS3100, for example, a reactivation fault takes 115μs. Hence, inappropriate tuning of the page-replacement algorithm can greatly decrease an application's performance.

Furthermore, an extremely large inactive queue increases the overhead of the pageout daemon. For each unreferenced page that it finds on the inactive queue, it must pass up some number of referenced pages. If the inactive queue contains most of physical memory and almost all pages on it are referenced, then the page-replacement algorithm's performance degenerates. To find an unreferenced page, the pageout daemon will search the large inactive queue, moving referenced pages from the inactive queue to the active queue and then back to the inactive queue to replenish it.

The solution to this dilemma in Mach 2.5 was to make it possible for machine-dependent code to override the boot-time configuration of `vm_page_inactive_target` and the other parameters of the page-replacement algorithm. The default size of the inactive queue was increased to two-thirds of available memory, and architectures without hardware reference support overrode this to use approximately 2% of available memory.

3 TLB-Based Reference Bit Emulation

Some architectures, like the MIPS architecture [Kane 88] used in the DECstation 3100, provide a TLB which requires software refill. Because the operating system supplies the TLB refill handler, these architectures present considerable flexibility for the operating system to tailor TLB refill to its needs. For example, TLB refill may be used to provide page-referenced information. We implemented this successfully in Mach, but at the cost of increased TLB miss latency.

We experimented with having the MIPS TLB miss handler emulate reference bits. When emulating reference bits, the MIPS pmap module maintains an array of reference bits, one per physical page. For efficiency, each reference bit is stored in a separate byte and a non-zero byte indicates the corresponding physical page has *not* been referenced. The pmap operations `pmap_clear_reference` and `pmap_is_referenced` index into the array and manipulate the appropriate byte. The TLB miss handler clears the byte for a physical page when loading a mapping for that page into the TLB. This modification to the miss handler increases the latency of servicing a miss from 9 cycles to 16 cycles.

With this technique, the MIPS architecture appears to support page-referenced information just like architectures with true support in hardware. The pmap interface hides the difference from the machine-independent page-replacement algorithm. Consequently, with reference bit emulation enabled the MIPS architecture can configure a large inactive queue and enjoy the benefits of an improved LRU approximation.

However, the latency of TLB refill is critical to overall system performance. For example, we counted the number of TLB misses generated by the compilation test of Section 2.3, run single-user on a DS3100 with 16 megabytes of memory and an instrumented kernel. In this environment, the compilation test has enough available memory to run to completion without page replacement. On average, the compilation test invoked the TLB miss handler 560,000 times. Given the 16.67Mhz clock rate of the DS3100 and the increase in latency of the miss handler, we can calculate

$$560000 * (16 - 9)/16.67\text{Mhz} = 0.23\text{s}$$

the performance degradation expected for reference bit emulation. To confirm this effect, we also timed the compilation test. With reference bit emulation, the test completed in 19.4s. Without reference bit emulation, the test completed in 19.2s.

Overall, reference bit emulation via TLB misses was an improvement because it let us increase the size of the inactive queue and reduce paging. However, this solution did not apply to other architectures lacking hardware page-referenced information, like the VAX, and it cost about 1% in overall performance when no paging was occurring.

4 Machine-Independent Reference Bit Emulation

The Mach 3.0 implementation of the FIFO with Second Chance page-replacement algorithm uses a machine-independent reference bit and reference faults to emulate hardware reference bits. With this modification, the algorithm operates similarly whether or not the hardware architecture supplies page-referenced information. Reactivation fault thrashing is no longer a concern.

The reference fault mechanism extends the machine-independent page structure with a reference bit. As before, when the pageout daemon moves a page from the active queue to the inactive queue, it uses `pmap_clear_reference` to clear machine-dependent page-referenced information, or remove all mappings for the page if the pmap module does not support reference information. At this time, the page's machine-independent reference bit is also cleared. When a fault on an inactive page occurs, the fault handler does *not* activate the page. Instead, it sets the machine-independent reference bit, creates a mapping for the page, and leaves the page on the inactive queue. These faults are known as *reference faults*. Before the pageout daemon moves a page from the inactive queue to the free queue, it uses the machine-independent reference bit and `pmap_is_referenced` to check both machine-independent and machine-dependent sources of page-referenced information. Figure 5 outlines the modified algorithm.

```
vm_pageout_scan() {
        while (vm_page_free_count < vm_page_free_target) {
                page = dequeue(vm_page_inactive_queue);
                if (page->referenced ||
                    pmap_is_referenced(page->phys_addr)) {
                        enqueue(vm_page_active_queue, page);
                } else {
                        clean the page if it is dirty;
                        enqueue(vm_page_free_queue, page);
                }
        }

        while (vm_page_inactive_count < vm_page_inactive_target) {
                page = dequeue(vm_page_active_queue);
                pmap_clear_reference(page->phys_addr);
                page->referenced = FALSE;
                enqueue(vm_page_inactive_queue);
        }
}
```

Figure 5: The Mach 3.0 Pageout Daemon

This modification unifies the operation of the page-replacement algorithm on architectures with and without page-referenced support. In both cases, references to pages on the inactive queue leave the page on the queue until the pageout daemon notices that the page has been referenced. In both cases, pages on the inactive queue may be mapped into address spaces. Therefore, the reference fault mechanism eliminates the need to tune the algorithm for different architectures. Machine-independent code can configure the size of the inactive queue.

Because pages on the inactive queue can be mapped into address spaces, architectures without page-referenced support no longer suffer from reactivation fault thrashing when the inactive queue is large. If the system's working set exceeds the size of the active queue but fits in memory, then the system reaches a quiescent state in which most of the inactive pages are marked as referenced and are mapped.

Allowing inactive pages to be mapped in effect allows the size of the inactive queue to change dynamically. When the system is paging heavily, few pages on the inactive queue are referenced. When the paging rate drops because the system's working set is resident, then most pages in the inactive queue become referenced and the effective size of the inactive queue decreases. The advantage of this method over explicitly adjusting `vm_page_inactive_target` for difference performance regimes is that the changes in the composition of the inactive queue happen automatically.

5 Page Replacement and External Memory Management

The pageout daemon faces two related problems, avoiding deadlocks and flow control, in addition to its primary problem of choosing pages to remove from main memory. The original page-replacement implementation proceeded the implementation of the external memory manager (XMM) interface [Rashid et al. 87, Young 89], and this greatly simplified the pageout daemon. The Mach 3.0 pageout daemon uses several strategies to cope with slow or uncooperative external memory managers.

When the original pageout daemon cleaned dirty pages, it copied them into the I/O system's buffer cache. The size of the cache automatically limited the rate at which the daemon could clean pages, because once it filled with data waiting to be written to disk, the daemon would start waiting for disk writes to complete and free buffers.

In contrast, the Mach 3.0 pageout daemon sends dirty pages to memory managers in `memory_object_data_write` messages. Because the XMM interface is asynchronous, the pageout daemon does not know when a memory manager has completed processing a `memory_object_data_write`. If the pageout daemon encounters a long sequence of dirty pages on the inactive queue and makes no attempt at flow control, it can generate arbitrarily long queues of `memory_object_data_write` messages waiting in the IPC system [Draves 90]. The normal queue-limiting flow control mechanisms of the IPC system are not appropriate (and are not applied) because the pageout daemon can't wait for a potentially buggy or malicious memory manager to take some action.

The Mach kernel does rely on a distinguished memory manager, the default memory manager [Golub & Draves 91], to operate correctly. The default memory manager services memory objects created when a process allocates temporary memory, not managed by an explicitly specified memory manager. The default memory manager also acts as the "pager of last resort," which cleans the dirty pages that unprivileged memory managers fail to process.

The pageout daemon and the default memory manager on occasion must allocate memory, consuming free pages, while they operate to create more free pages. For example, the pageout daemon allocates message buffers for `memory_object_data_write` messages. To prevent a deadlock when there are no free pages, the kernel maintains a reserve of free pages and only the pageout daemon and default memory manager can allocate free pages from the reserve. If the pageout daemon fails at flow control and generates `memory_object_data_write` messages more quickly than they are processed, then it can consume the free page reserve and deadlock.

The Mach 3.0 kernel uses three thresholds to prevent the free page queue from being exhausted:

`vm_page_free_reserved`
: Below this level, only privileged threads can allocate free pages.

`vm_pageout_reserved_internal`
: Below this level, the pageout daemon no longer expects memory managers other than the default memory manager to function. The pageout daemon cleans pages associated with other memory managers by double-paging them, or sending them first to their real memory manager and then immediately resending them to the default memory manager.

`vm_pageout_reserved_really`
: Below this level, the pageout daemon stops operating and waits for the default memory manager to catch up.

Artificially constructed paging test programs can drive the size of free queue below the `vm_pageout_reserved_really` threshold, but the `vm_pageout_reserved_internal` threshold is rarely exceeded in practice.

The Mach 3.0 pageout daemon uses two strategies for flow control, to control separately the rate at which dirty pages are sent to the default memory manager and to other memory managers. Because of the constraints placed on the default memory manager, that it must be correct and it must free pages after cleaning them, the pageout daemon can use a reliable strategy with it. The strategy employed with other memory managers is only heuristic, but the `vm_pageout_reserved_internal` threshold described earlier prevents this from compromising system correctness.

To avoid flooding the default memory manager, the kernel keeps a "laundry count" of how many pages the default memory manager has not yet cleaned. Pages sent to the default memory manager are marked with a "laundry bit," and when pages so marked are returned to the free queue, the laundry count is decremented. The pageout daemon pauses to let the default memory manager catch up when the laundry count exceeds a threshold, `vm_pageout_burst_max`.

To avoid flooding other memory managers, the pageout daemon also pauses after sending out `vm_pageout_burst_max` dirty pages. Because the kernel does not have an equivalent of the laundry count for these memory managers, the pageout daemon must assume that its pause gave the memory managers time to process the `memory_object_data_write` requests.

The two flow control strategies work best when the pageout daemon's pauses are properly tuned. Instead of requiring machine-dependent tuning, the pageout daemon tunes the pause interval dynamically. If the pageout daemon wakes from a pause and the laundry count is still high, the pageout daemon increases the pause interval. If the laundry count is low after each of several consecutive pauses, the pageout daemon decreases the pause interval.

6 Related Work

Other operating systems for the VAX architecture, notably VMS and BSD Unix, perform page-replacement without the benefit of hardware page-referenced information. Some researchers have taken a different approach to page replacement in Mach and have experimented with application-level control over page-replacement decisions.

The VAX/VMS operating system[Goldenberg & Kenah 91, Turner & Levy 81] uses per-process working sets with global free and modified lists to control page replacement. Normal FIFO page replacement occurs within a working set, each of which behaves like Mach's active queue. The global free and modified lists cache pages not mapped into processes. Page faults which are satisfied from the free and modified lists without I/O are known as *soft page faults*. Like Mach's reactivation faults, they ameliorate the FIFO replacement within working sets to produce an LRU approximation. They also suffer from the same performance tuning dilemma. For example, [Lazowska 79] found that the default size for the free and modified lists was too small for memory-poor environments.

The BSD Unix operating system uses a clock algorithm [Babaoglu & Joy 81]. The clock hand cycles around main memory, clearing software reference bits and unmapping pages. Faults on unmapped pages, like Mach's reference faults, set the software reference bit. The clock algorithm reclaims unreferenced pages found by the hand. Unlike Mach's algorithm, the BSD algorithm suffers from scalability problems, because the clock hand must sweep across the machine's entire physical memory. It is also difficult to tune properly the speed of the clock hand.

Some researchers [McNamee & Armstrong 90] have taken the approach of providing control over page-replacement to the application. This is a promising approach for applications like Lisp, ML, and databases, but it leaves open the question of a default mechanism. Mach's FIFO with Second Chance algorithm can easily accommodate simple application-level hints about the relative worth of pages. Pages deemed less valuable can be forcibly moved from the active queue to the inactive queue to make them more likely candidates for replacement.

7 Conclusions

The Mach kernel uses a FIFO with Second Chance page-replacement algorithm. We have experimented with several approaches to providing the page-referenced information that the algorithm needs to approximate LRU behavior. An approach based on a machine-independent reference bit and reference faults outperforms Mach 2.5's reactivation fault technique and reference bits based on software TLB refill. The FIFO with Second Chance algorithm can accommodate the requirements of external memory management.

Acknowledgements

Avadis Tevanian and Michael Young first implemented page-replacement in the Mach kernel. Alessandro Forin implemented the TLB-based reference bit emulation described in Section 3. Michael Young implemented preliminary versions of the mechanisms described in Section 5.

References

[Babaoglu & Joy 81] Babaoglu, O. and Joy, W. Converting a Swap-Based System to do Paging in an Architecture Lacking Page-Referenced Bits. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles*, pages 78–86, December 1981.

[Belady 66] Belady, L. A. A Study of Replacement Algorithms for a Virtual-Storage Computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[DEC 79] Digital Equipment Corporation. *VAX-11 Architecture Handbook*, 1979.

[Denning 68] Denning, P. J. The Working Set Model for Program Behavior. *Communications of the ACM*, 11(5):323–333, May 1968.

[Draves 90] Draves, R. P. A Revised IPC Interface. In *Proceedings of the USENIX Mach Workshop*, pages 101–121, October 1990.

[Goldenberg & Kenah 91] Goldenberg, R. E. and Kenah, L. J. *VAX/VMS Internals and Data Structures: Version 5.2.* Digital Press, 1991.

[Golub & Draves 91] Golub, D. B. and Draves, R. P. Moving the Default Memory Manager out of the Mach Kernel. In *Proceedings of the Second USENIX Mach Symposium*, November 1991. This issue.

[IBM 88] International Business Machines, Austin, Texas. *IBM RT PC Hardware Technical Reference Volume 1*, third edition, 1988.

[Kane 88] Kane, G. *MIPS RISC Architecture.* Prentice-Hall, Englewood Cliffs, NJ, 1988.

[Lazowska 79] Lazowska, E. D. The Benchmarking, Tuning and Analytic Modeling of VAX/VMS. In *Papers Presented at the 1979 Conference on Simulation, Measurement and Modeling of Computer Systems*, pages 57–64, August 1979.

[McNamee & Armstrong 90] McNamee, D. and Armstrong, K. Extending the Mach External Pager Interface to Accommodate User-Level Page Replacement Policies. In *Proceedings of the USENIX Mach Workshop*, pages 17–29, October 1990.

[Rashid et al. 87] Rashid, R., Tevanian, Jr., A., Young, M., Golub, D., Baron, R., Black, D., Bolosky, W., and Chew, J. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 31–39, October 1987.

[Tevanian 87] Tevanian, Jr., A. *Architecture-Independent Virtual Memory Management for Parallel and Distributed Environments: The Mach Approach.* PhD dissertation, Carnegie Mellon University, December 1987.

[Turner & Levy 81] Turner, R. and Levy, H. Segmented FIFO Page Replacement. In *Proceedings of the Conference on Measurement and Modeling of Computer Systems*, pages 48–51, September 1981.

[Young 89] Young, M. W. *Exporting a User Interface to Memory Management from a Communication-Oriented Operating System.* PhD dissertation, Carnegie Mellon University, November 1989.