Better Control Over Memory Management: Hinting with Love/Hate

J. Neubert D. Gekiere
Computer Science,
University of Wisconsin
Madison, WI, USA 53706
neubert@robios6.me.wisc.edu gekiere@cs.wisc.edu

Abstract

Current memory management policy is, on most systems, global and inflexible. Current systems provide poor mechanisms to allow the programmer to share knowledge of access patterns that may generate improved page swapping and caching. In this paper we modify a popular operating system's memory management policy to take into account hints given by the programmer. The system consists of love and hate hints given to the kernel via system calls. It gives the programmer the option of moving a page to the front of the swap line or informing the operating system that the page should be kept on the active list. The paper demonstrates the improved performance that a system can gain through the use of the love and hate hints.

Keywords: Memory Management, OS Hints

1 Introduction

Memory management presents a rather difficult optimization problem with no best solution. Current systems tend to use a global policy needs to balance factors like the amount of state, the type of policy (i.e. LRU, MRU), hardware, and system complexity. The designer attempts to produce a system that performs well in most situations. A more optimal policy may be created if the operating system (OS) can obtain knowledge from the applications. In this paper we will propose a mechanism to allow applications to share their knowledge with the global policy in a attempt to decrease the paging in the system.

Some modern operating systems such as Unix and Linux provide a madvise system call. The system call allows the user to specify the way that the memory will be accessed (or not accessed for that matter). The OS then assigns the memory the policy it thinks best fits the hint the user supplied. This system does not allow for the kind of flexibility that an application may need, performance wise.

Unix and Windows 2000 provide the ability to lock pages into memory. This prevents the pages from being swapped out altogether. If the memory is needed and the only way to acquire it is to remove a locked page from memory, the entire process that holds the locked page is removed from memory and stored on disk. The process will only be run after enough memory becomes available. This can lead to an excessive amount of paging especially in cases where the computer's memory is limited, and more importantly it can stop the application dead in its tracks.

Micro-kernels like Mach [8] and Exokernel [4] have purposed that each application provide its own swapping policy or accept a policy provided by a user library. The difficulty with these systems is that it is a rather involved process to write a swapper. A user may be interested in what happens to a small percentage of the pages her application is using, and might not want to provide a complete swapping policy - it simply is not a cost effective solution. Micro-kernels also tend to have a rather large overhead because these self-implemented policies occur at user (as opposed to kernel) level, which can generate an excessive amount of kernel/user interaction.

An adaptive system could be required to learn from the memory access patterns of the applications running on top of it. Generic, fixed global policies could evolve according to the needs of these applications. This would likely involve complicated prediction and learning techniques as an application's behavior will likely evolve over time. It would also require both time and space overhead (disk and memory space to trace behavior, and CPU time for analysis). Work on adaptive caching has been done, but in the context of distributed memory architectures [1], and distributed systems [2].

In this paper we propose using a modified version of a global, fixed memory management policy. This pa-

per purposes a simple hinting mechanism. The hints provide a conduit for the programmer to influence the decisions made by the swapping daemon. This has several advantages over the policies discussed above. Unlike the Windows 2000 or Unix systems, which offer only a lock, the system presented in this paper provides 255 hinting levels with the maximum level similar to locking a page. Also our system does not require that an entire process be swapped out if a locked page is swapped out. The hinting system also provides users with the flexibility of using hints on as many pages as desired (or none at all), without having to implement an entire swapping system. The hinting system adds a minor overhead cost to the present system requiring only two system calls be implemented and small modifications to the system's swap daemon be made.

The outline of the paper is as follows. Section 2 briefly describes the idea behind love/hate hints. Section 3 describes the current Linux memory management system. Section 4 explains the different changes we made to this system in order to implement hinting. Section 5 describes the experiments we used to test our modified system and measure performance improvements. Related works are presented in section 6. Finally, the paper in summarized in section 7.

2 The Love/Hate System

Most memory management systems attempt to evict the LRU (least recently used) pages when there are processes contending for memory. The love/hate mechanism is intended to manipulate the order of the pages. This system supplies two system calls: love and hate. When love is called on a region of memory the pages containing the memory will de facto become the most recently used. Conversely the hate call manipulates the memory management system into believing the page or region of pages are the least recently used. Hated pages are then most likely to be swapped to the backing store. This would allow a program to emulate a MRU (most recently used) memory management policy. Hating every page after it is accessed so that the most recently accessed page is the first to be evicted when space is needed.

3 Linux Memory Management

In order to understand how the Linux 2.4.19 kernel was modified, a basic understanding of the Linux memory management system is required. The current Linux system is based on the clock algorithm. It maintains three doubly linked lists of pages: a free, an inactive, and an active list. The movement ratio between the lists are facilitated by the kswapd daemon. Figure 3 shows the lists and the functions in the daemon responsible for the movement of the pages between the lists.

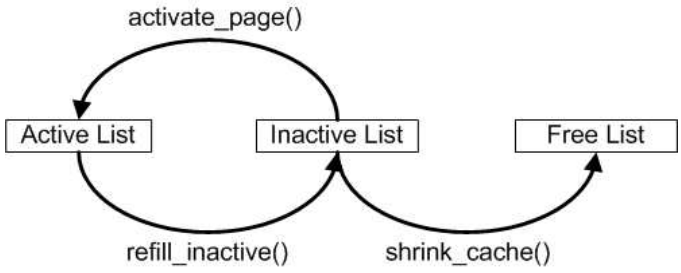


Figure 1: The three lists that the kswapd daemon uses to manage the memory in the Linux 2.4 kernel. The functions used by kswapd are listed in the figure above and below the arrows, which indicate the movements that they facilitate.

The kswapd daemon uses these functions to keep the lists in balance. The balance is achieved by maintaining an approximately two to one ratio between the number of active pages and that of inactive pages. The kswap daemon also attempts to keep a minimum number of pages in the free list so that an allocation request can be handled quickly. When the lists are out of balance the kswapd daemon is awaken more frequently until balance is achieved.

The modifications outlined in this paper will focus primarily on the movement of pages from the active list to the inactive list, which is facilitated by the refill_inactive function (See Figure 3). This function scans the active list examining the state of each page. The state of a page is stored in a page structure(See Nayani [5]). This structure contains a reference bit that is set when the page has been referenced. The function tests and clears the reference bit. If the reference bit was set the page is allowed to remain on the active list, otherwise it is removed and placed on the inactive list.

4 Methodology

Here we are going to describe how the love/hate system (described in section 2) is implemented in the Linux memory management system (described in section 3). The system required the addition of two system calls and some small modifications to the kernel's swap daemon.

The love/hate system described above simply moves

the pages to the from or back of the LRU queue. Unfortunately Linux does not implement a strict LRU queue, but rather a approximation to a LRU. The lists are circular and the daemon simply moves around it storing the point at which it stopped internally. This makes it nearly impossible to implement the system as it was outlined above.

4.1 Initial System

The initial love/hate system was rather simple. The love call moved pages to the active list and the hated pages to the inactive list. This system was found to be flawed. Hated pages frequently moved right back to the active list after they were hated. At first it was thought that the software reference bit was causing this to occur, but that was not the case. Even when the reference bit in the page structure was cleared the pages still seemed to find their way to the active list after being hated. The only way to prevent this was to clear the software bit as well as the hardware bit. This led to a second generation of the system calls that not only moved the pages, but modified their state as well. In the case of hated pages the reference bits were cleared as mentioned above and loved pages had their reference bits set.

The initial system proved functional. It moved pages to the appropriate lists, but the effect was rather short-lived. This wasn't as big of an issue for hated pages, but might have been for pages that a programmer felt were particularly important. After just two scans of the memory by the swapping daemon the page could be ready for swapping. The only way to keep them from moving to the inactive list and being swapped was the use of repeated and frequent love calls. This was rather costly in terms of performance.

4.2 Current System

The initial system did prove functional, but lacked the lasting effect and flexibility we desired. The benefits of the original system were that is was stateless and simple. It also provided anecdotal evidence that the system could provide improved performance. The system that is outlined here builds on the successes of the first system and addresses the problems mentioned above.

The new love call had state associated with it, unlike the initial one. The page structure referred to in section 3 was modified to contain an unsigned character, love. This was used to store the level of love a page has. The addition of the love level to the page state addressed the transient nature of the initial love call, removing the need for frequent love calls.

The love level can be used to provide an approximate ordering to the loved pages. Love will prevent the page from moving to the inactive list in the memory structure. Every time love is used to prevent the movement of a page it is decremented. The page will not be removed from the system until the page's love is exhausted and it is moved to the inactive list. These modifications required only 6 lines of additional code in the page daemon.

This treatment of love does not provide a strict ordering of the pages, but rather a hint to the system of the priority of the page. This was done for several reasons. If the pages are provided a strict priority it is difficult to determine how one might handle the memory of a non-hinting application. A non-hinting application should not be unjustly punished because it does not love its pages. Also if one were to use a strict ordering the page daemon would have to be changed significantly so it would search for and remove only those pages with the lowest love level.

The fading love aids in the prevention of thrashing that may be generated by a love hint. When in memory the number of free pages becomes low the paging daemon will run frequently, decrementing the love of unreferenced pages until they can be swapped out. This system will usually remove those pages with the lowest priority first, but frequently accessed low priority pages may not be removed. This behavior is aimed at preventing the misuse of love. An application that loves all its pages will have those that it is not referencing eventually removed.

4.3 The System Calls

The two system calls that were created can be seen in figure 4.3. Both the love and the hate calls require a beginning address and the amount of memory that the user wishes to hate or love. The love call requires the level of love that the user wishes to bestow on the region as well. The maximum amount of love is 255, which will lock the page in. All other levels are continually decrased by kswapd. Both system calls return the amount of memory that was loved, or hated.

```
unsigned long love( void* addr, unsigned long length,
                                        unsigned char love);

unsigned long hate( void* addr, unsigned long length);
```

Figure 2: The Love/Hate Hint API

One of the most difficult issues to deal with was to

decide what to do about pages that are invalid. In the case of hate it is rather easy. We just ignore the page and move on to the next. The love call was more difficult. The question that needed to be answered was: should the page be fetched or created? If the love call did create and fetch invalid pages in the region to be loved it would enable misuse and increase the likelihood of accidental love. A love call that brings loved pages into physical memory would also need to be limited so that a user could not love more pages than are available. This leads to the decision that despite the advantages of a system like madvise, which prefetches pages, the love call would not fetch the pages. Instead the love call exits and returns the amount of memory loved (so the programmer knows that the pages that have effectively been loved are sequential in memory, starting at the address he or she provided). If desired, the programmer can then touch or fetch the pages that where not in physical memory and call love again on those pages.

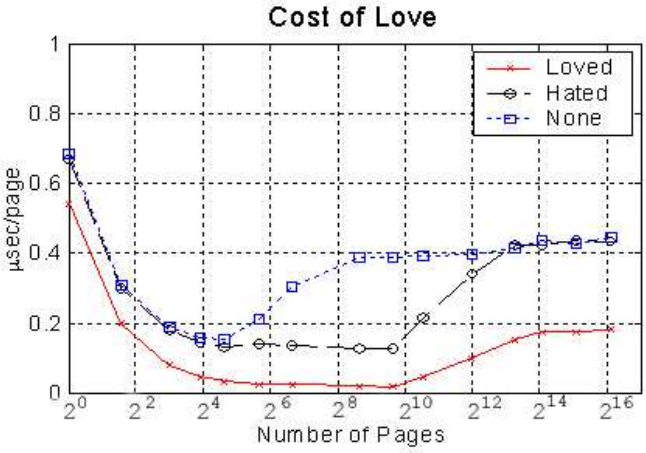


Figure 3: The graph has three series of data displaying the amount of time per page of a love call. The loved series refers to pages that were loved prior to the love call. The hated series refers to pages that were hated prior to the love call. The last series, none, refers to pages that were neither hated nor loved.

5 Experiments

In order to evaluate the performance of the love/hate hinting system, the cost per page of using hate and love is quantified. Then the system is used by an application which attempts to mosaic several images. Another set of experiments looks at typical databases I/O patterns, and tries to use love and hate hints to simulate both LRU and MRU policies on simultaneous file accesses.

5.1 The Cost of Love and Hate

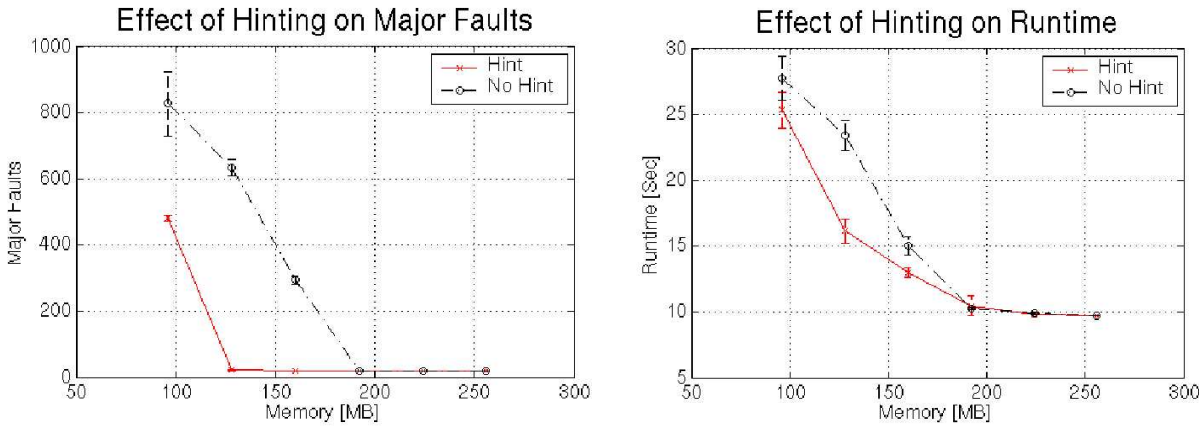
The cost of the system was the first thing that was measured . The testing was done on a Pentium III 800 MHz system with 384 MB of ram. The page size is 4 KB. The system had all non-essential processes killed to avoid anomalies in the measurements. This system was used to produce all the data seen in figures 3 and 4. The graphs show the cost of loving and hating pages with various size calls. The cost is given per page because the operation is page-wise.

The cost of hating previously loved or hated pages was much higher despite the fact that the operations were rather similar. The cost of hating pages that were loved was about 30% more then a page that is already hated. This is likely due to the movement of the pages from the active list to the inactive list. Conversely the cost of loving a page that is already loved is almost non-existent with respect to loving a hated page. The cost of the initial call to love or hate was nearly identical.

The most interesting feature of the data was the large dip in the graph. The large initial cost is due to plain system call overhead, generated by crossing into the OS. As the system call is used on larger ranges of pages the cost is amortized. The low portion of the dip is likely where the L1 cache is being used. Eventually the system must go to main memory and the cost per page reaches its steady state cost. So when love or hate are called on a region of memory, some data must get cached. It is important to note that every step was taken to prevent caching of data on the processor from affecting the results, including accessing several megabytes of memory, but the low portion of the plot remained significantly longer than that of the region of memory with no prior calls made on it.

5.2 Image Mosaic Benchmark

Applications which perform a image mosaic operations require large amounts of memory and are sensitive to memory management policies. The benchmark creates a 72 megabyte (MB) aggregate image structure along with about 144 MB of loaded and processed images. Many of the processed images, while not needed in the application after the initial use, can not be deleted because of manual tuning, which uses the data.



(a) (b)

Figure 5: In the above figures each data point is the average of 12 data points. The whiskers on the point represent two standard deviations on either side of the average. (a) shows the number of major page faults that was generated by the image mosaic benchmark both with and without the use of hints. Major faults refers to the act of moving pages from the swap area back into main memory. (b) indicates the runtime of the program both with and without hints. The memory axis indicates the physical memory the system had available.

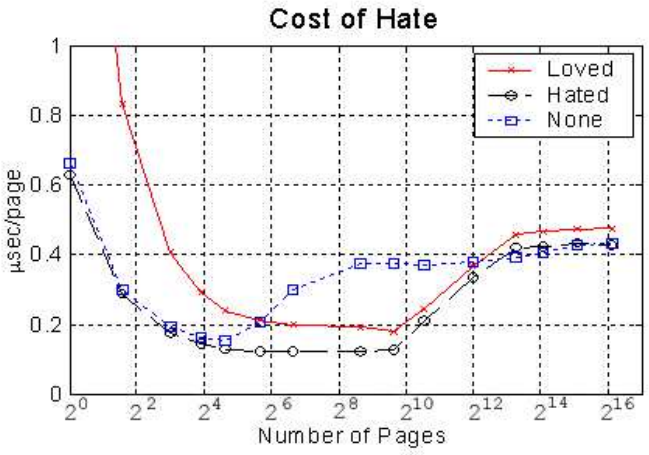


Figure 4: This plot outlines the cost of love per page with various sizes of memory. The plot is in the same format as that of figure 4.

The benchmark was run on the same Pentium III machine described above with varying amounts of physical memory.

One of the problems that was experienced during the addition of hints was that they need to be used in an intelligent manner. When they were added initially whole images were loved with the same amount of love. This worked well as long as the system had 128 MB or more of main memory available. The hints even improved the performance for all but the 96 MB point. The hints actually severely hurt performance at the 96 MB level, probably because love was used with such broad strokes. This generated an excess of page faults because when a page that was needed was swapped in, a page that would be needed later was moved to disk. In order to get the best performance the love should be used in a intelligent manner. This required varying the level of love on the aggregate image so that a large portion remains in memory at all time, but the remaining must be swapped in and out of memory.

The graph in figure 5(a) indicates that the application with the addition of hints can avoid many of the major page faults (Major page faults are generated when data on the disk is moved to main memory). This can generate additional runtime as can be seen in figure 5(b). While relationship can be observed between the number of major page faults and runtime, the system suffered some performance loss (regardless of the number of major page faults) as the memory decreased. This was likely due to the increased overhead of managing a tighter memory space. An important observation is that as memory is increased there is no notable cost to the use of hints, though non-hinting system does enjoy a slightly better performance when the system has more than 200 MB of memory, but the improvement is not significanti, and can be attributed

	Type of Hints			
Pass	None	Hate	Love	Both
#1	97	96	85	84
#2	41	59	51	32
Total	138	155	136	116

Table 1: Cold Cache Execution Time in Seconds

	Type of Hints			
Pass	None	Hate	Love	Both
#1	6	6	7	7
#2	32	4	17	5
Total	38	10	24	12

Table 2: Warm Cache Execution Time in Seconds

to the cost of love and hate.

5.3 Database Experiments

To test the usefulness ond usability of the love/hate hint system, we chose to try and reproduce some well-known repeated file access pattern. Databases file access behavior is predictable and depends upon the kind of request (or query) that is being made by the user. Dewitt and Chou [3] have catalogued those access patterns as the QLSM (Query Locality Set Model). Reference patterns are categorized in three ways: sequential, random, and hierarchical. Sequential accesses include straight sequential, clustered sequential (parts of the file might be read repeatedly), and looping sequential (the whole file is read more than once in its entirety). Random accesses include independant random (truly random) and clustered random (where a pattern of locality may appear). Hierarchical accesses mostly describe index tree traversals: straight traversal (from root to leaf), hierarchical with straight sequential (straight traversal followed by sequential scan), hierarchical with clustered sequential (traversal followed by clustered scan), and looping hierarchical (repeated traversal).

The following experiment tries to simulate a simple join operation. A join query builds a new table using attributes from two or more existing tables, selecting attributes according to one or several predicates. Our simulation will use two 'dummy' tables, and is only concerned with access to these tables . The creation of the new table is not simulated, since it does not lend itself to I/O optimization (either it is written to the disk, or not).

Set-up: The tables are 64MB each in size. We operate with a Linux station whose memory has been limited to 128MB. This allows the characteristics of the memory management policy to be observed. A portion of the 128 MB of main memory is required by the kernel and various processes, thus only a portion of the 128 MB of the files can be cached in memory. This generates swapping, which demonstrates the policy performance.

One of the files is considered the outer file, the other one the inner file. For each page of the outer file, a page of the inner file is read as well. The access models chosen for this experiment are looping sequential for the outer file, and independent random for the inner file. The loop runs twice on the outer file. To ensure cache freshness, a 128MB dummy file (different of course from the ones used as tables) is mmap-ed and accessed sequentially prior to starting the actual measurements. Two sets of measurements were taken, one with a 'cold' cache, i.e. where the application simply gets the data from the mmap-ed files as it goes, and one with a 'warm' cache, where the application does a sequential read of the inner file prior to starting the measurements.

Our goal was to simulate MRU page replacement policy for the outer file, and LRU policy for the inner file. These page replacement policies make sense for the access patterns that we are dealing with here. MRU makes sense for a file that is sequentially accessed repeatedly, because once a page is read, it won't be needed again until we cycle through the whole file. LRU makes sense for random access, because we want to keep as much of the file in memory as possible (hence the cache with the inner file).

For each set of measurement, love and hate hints were administered as follows: none (no hints given), hate only (hate hints given to a group of 64 previously read pages on the outer file), love only (in the cold cache measurements, love hints are given to each inner file page that is accessed in the loop; in the warm cache measurements, a general love hint is given to the whole mmap-ed file once it has been read into memory), and love and hate hints at the same time (as described previously). The number of pages on the active and inactive lists was taken at the very start, after pass #1, and after pass #2. Also, for the warm cache experiments, after the cache was warmed (and 'loved', if that hinting took place). Both passes were timed, and totaled.

The best results, performance-wise, come from the warm cache experiments (See table 2), as should be expected, but the cold cache experiment's results provided insight into the effectiveness of different strategies for using the love/hate hints. A simple cache warming with no hints already shows considerable improvement over any of its counterparts (38 seconds vs.

	Type of Hints							
	None		Hate		Love		Both	
Page Type	Inactive	Active	Inactive	Active	Inactive	Active	Inactive	Active
Starting Point	17864	12380	16659	13508	15001	14928	18416	11694
Cache Warmed	21256	8246	20671	8793	4016	25268	5095	24211
After Pass #1	7924	22348	23808	5874	12937	16984	18455	11638
After Pass #2	8005	21831	15520	14773	11873	17576	16284	13945

Table 3: Warm Cache List Sizes

	Type of Hints							
	None		Hate		Love		Both	
Page Type	Inactive	Active	Inactive	Active	Inactive	Active	Inactive	Active
Starting Point	14661	15539	14471	15761	18442	11642	18274	11788
After Pass #1	12714	16843	24691	4992	19098	10158	19099	10151
After Pass #2	6830	22892	16065	13648	16044	13208	16899	12369

Table 4: Cold Cache List Sizes

116 seconds for the best cold cache result). A love hint on the inner file improves this figure by about 35% (24 seconds). This last result is improved by the use of hate hints on the outer file (in conjuntion or not with love hints on the inner file). It seems hinting hate alone (no love) gives this set-up a slight edge over its 'loved & hated' counterpart (10 seconds vs. 12 seconds). This might be due to the overhead of treating an active list with many loved pages. Hence in a moderately memory-pressured environment, a cache warming on data that is often used might suffice to the application's needs, if it is used in conjuntion with hate hints on data that can be swapped out immediately (MRU style). In our case the hate hints on the outer file make Linux MM system reuse those pages assigned to the mmap-ed file as it is being read, in effect keeping the pages of the inner file in memory.

The cold cache experiments ran much slower, but hints improved the situation somewhat. The surprise here came from the 'hate only' hinting experiment, which ran slower than no hint at all (155 seconds vs. 136 seconds). In this case it seems the overhead of hate, moving the pages from the active to the inactive list, actually gets in the way of the rest of the system. The best results came from using both love and hate hints, and the effects are clear when one looks at the individual passes execution times; the second pass in that case is faster by almost 25% compared to the second fastest (32 seconds vs. 41 seconds - no hints). The ever-synergetic work of love and hate accomplished during the first pass obviously pays off during the second pass.

Tables 3 and 4 detail the inactive and active list sizes during the experiments. As expected, the active size list grows with love calls (table 3, look at the difference between 'Starting Point' and 'Cache Warm' when a love hint is provided). Similarly, the inactive list size grows when hate is provided (in both tables, for the 'hate' only experiments, look at the difference between Pass #1 and Starting Points sizes). These measurements are consistent with the behavior that was expected from the love/hate system, and show that two different replacement policies (MRU and LRU) can be simulated at the same time within an application.

6 Background

Most of the work on hinting has centered around file prefetching. Notably, Patterson and Gibson [7] describes the TIP system (Transparent Informed Prefetching). The authors observe the usual I/O / CPU performance gap, and state their goal of reducing read latency. The TIP system is focused on prefetching data before it is needed by the application (contrast with our system, where we are concerned about data that is already present in memory). TIP is based on the application's knowledge of future I/O accesses, therefore hints are expressed in terms of operations on files. These hints are used not only to prefetch the data, but also to optimize I/O access, using low level knowledge of the system (seek schedule optimization, more efficient utilization of disk array if present, etc - details that the programmer does not and should not possess). The hints are described as 'disclosure', as opposed to 'advice', because they inform the system of actual future operation on files. This should be contrasted with our love/hate hints, which are in

effect 'advice' for the memory management system. A love hint does not guarantee that the targeted page or memory region will effectively be in pysical memory when needed (though it does, of course, make it extremely likely).

The TIP system was targeted at a single application running on a computer host. A follow-up system, TIP-2 [6], exploits hints for both file prefetching and informed caching (TIP-2's cost estimators decide which block to eject from the cache). This was extended to 'TIPTOE' [9] for multiple processes running and giving hints concurrently.

7 Conclusion

We have presented a modification to a standard memory management system, that allows application programmers to give hints to the underlying system as to which pages should be kept in physical memory, and which pages should be potentially disposed of. Our love/hate system is not completely transparent (the programmer gives explicit hints), but is very much unobtrusive; two simple system calls can be used to add love/hate hints, but their use is not mandatory, and not using them yields standard behavior.

Through various experiments, we have showed that the hints improve overall applications performance, though indiscriminate use of both kinds of hints is not recommended - i.e. each application should evaluate its own needs.

Our hints can be used to simply improve performance on memory demanding applications, and simultaneously simulate different memory management policies. Other possible uses could include file prefetching, through the use of threads (a thread reads and loves different files that are going to be needed by the application later on).

Remember: all your applications need is a little love.

8 Acknowledgements

We would like to thank Professor Remzi Arpaci-Dusseau for his help.

References

[1] J. Bennett, J. Carter, and W. Zwaenepoel. Adaptive software cache management for distributed shared memory architectures. In *Proc. of the 17th Annual Int'l Symp. on Computer Architecture (ISCA'90)*, pages 125–135, 1990.

[2] P. Carns, W. Ligon III, R. Ross, and R. Thakur. Parallel file system for linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, 2000. USENIX Association.

[3] H. Chou and D. DeWitt. An evaluation of buffer management strategies for relational database systems. In Alain Pirotte and Yannis Vassiliou, editors, *Proceedings of 11th International Conference on Very Large Data Bases*, pages 127–141. Morgan Kaufmann, 1985.

[4] D. Engler, M. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Symposium on Operating Systems Principles*, pages 251–266, 1995.

[5] A. Nayani. Linux memory management, April 2002.

[6] R. Patterson, G. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *In Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 79–95, December 1995.

[7] R. Patterson, G. Gibson, and M. Satyanarayanan. A status report on research in transparent informed prefetching. *ACM Operating Systems Review*, 27(2):21–34, 1993.

[8] R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew. 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, Palo Alto, California, 1987.

[9] A. Tomkins, R. Patterson, and G. Gibson. Informed multi-process prefetching and caching. In *Proceedings of the 1997 Conference on Measurement and Modeling of Computer Systems*, pages 100–114. ACM Press, 1997.