Application Specific Memory Management

Shawn Chang, Kirby Zhang
Electrical Engineering and Computer Science
University of California, Berkeley
{shawnc, silkworm}@cs.berkeley.edu

Abstract

Different applications have different memory reference patterns. Most operating systems, however, are oblivious to this simple truth. Some algorithms are highly suitable for certain kinds of applications but inadequate for other kinds of workloads. We believe there is a range of applications that can benefit from application-specific memory management policies. Further, we observe that reference patterns do change during process execution, suggesting that applications can further benefit from dynamic selection of a suitable VM policy.

To demonstrate these ideas, we have implemented an application specific page replacement management system. Multiple page replacement algorithms co-exist in the kernel and each process can be assigned a better-than-default algorithm. Alternatively, a process can take advantage of the Dynamic Intelligent Algorithm Selection (DIAS) mechanism, which makes an attempt to select a suitable page replacement policy based on page fault history information. Our implementation includes a memory reference profiling system and a number of data analysis tools to help users make algorithm selections. We present performance data to demonstrate that for some benchmarks, an alternative algorithm offered by our system outperformed the Linux default by a considerable margin. Initial testing shows that dynamically selecting between two algorithms can give better performance than using either of the algorithms alone.

1 Introduction

General purpose operating systems choose virtual memory management policies that are thought to perform well for the intended range of applications. But memory reference patterns differ greatly from one application to another. Operating system designers are often forced to choose between applications that have diabolically opposing needs. The end result is that some "minority" applications are not well served by generic designs.

It is well known to the OS community that some rarely used algorithms are highly suitable for a narrow range of applications. While they do not perform well in the aggregate, they can excel for a specific access pattern. In cases of sequential access or random access to pages which will not be referenced in the near future [Stonebraker 81], MRU performs much better than the common LRU algorithm.

Insight: virtual memory management policies can be used in such a way that their effects are localized in both the dimension of applications and in the dimension of time. In other words, a management policy only needs to affect a specific set of applications during some interval of time. The portent is rather obvious: choose the policy that is best for a particular application during a particular time. Thus, an optimized virtual memory management policy can be chosen for a specific reference pattern.

In our work, we chose to focus on page replacement algorithms to demonstrate that selective deployment of a virtual memory policy can be done, and that with proper heuristics and tuning, it can be beneficial. We wanted to modify an operating system that 1) is modifiable (we must have access to the source code), and 2) has a large user population who may benefit from our work. Predictably, we chose Linux on the x86.

We have implemented a virtual memory system we call SM (let's call it Sensational Memory here, although the code name had somewhat different roots). SM adds the following features to Linux kernel 2.0.30 from a Red Hat distribution:

?? Reference and page fault pattern profiling for a single application for an arbitrary length of time (subject to storage constraints) at an arbitrary resolution (subject to performance constraints)

?? Through the use of a system registry, users can specify a suitable page replacement algorithm for any application. The specified algorithm will be applied within the virtual address space of that application. Interference with other applications running concurrently on the system will be minimal, but not non-existent.

?? To take advantage of the above flexibility, we implemented an algorithm that dynamically chooses one of n page replacement algorithms for a given application. We call it Dynamic Intelligent Algorithm Selection (DIAS).

To help ourselves as well as potential future users of our system decide on the appropriate pairings between application and algorithm, we also created the following tools:

?? A reference pattern playback engine. The tool takes profile data for an application and attempts to mimic its memory reference pattern. The playback engine is used to analyze application performance under different page replacement algorithms. It reduces our need to spend time waiting on non-memory accessing computation. With the playback engine, interactive applications can be profiled for extended periods and re-analyzed quickly.

?? A page fault rate analysis and reference pattern plotting tool. The visual displays aided us tremendously in our analysis.
?? A program that traverses our profile database and generates statistics reports on page fault rate.

Our initial performance results are promising. For some applications, we were able to choose a better-than-default algorithm without degrading performance elsewhere. By repeatedly serializing the execution of different applications, DIAS produced better performance than any single page replacement algorithm.

The next section discusses the SM architecture. Section 3 describes DIAS in detail. Section 4 gives an overview of the five page replacement algorithms implemented for SM. We analyze performance data we have collected in section 5. Section 6 discusses related work. Section 7 explores future work. Section 8 describes our experience with Linux and the development of SM. We conclude in Section 9.

2 Architecture

In designing the architecture of SM, we kept a number of goals in mind. We did not want to disturb the interface between applications and kernel services. We wanted to minimize the performance impact of algorithm multiplexing as well as application profiling. To the extent possible, we wanted to localize the effect of a page replacement policy to the intended domain. Finally, we wanted to minimize the need for system reboots during development and administration. Without a good design in meeting the last goal, we would never have got as far as we did.

2.1 Profiling

Profiling activity is specified in the system resource file. When the file is modified, SM Daemon (SMD) detects the change and rereads the file. To minimize the performance impact, we decided that only one application will be profiled at a time.

After several design iterations, we arrived at a non-obtrusive, low overhead solution to reference pattern profiling. SMD was created as a kernel thread that periodically sweeps through physical memory pages allocated to the application under profile. SMD reads the *referenced* bit and *page_faulted* bit for each page table entry (PTE) and writes that information to the profile log. The *referenced* bit is set by the 386 MMU during every page reference. The *page_faulted* bit is originally an unused bit in the page table entry. We made a small modification in the page fault interrupt handler to set this bit at each page fault. After writing to the log, SMD clears both bits. Thus, *referenced* bit tells us if the page was referenced since the last sweep, and the *page_faulted* bit tells us if a reference to that page resulted in a page fault.

The *referenced* bit, however, is also used by most page replacement algorithms, which read and clear this bit to obtain age information. To work around this, we created a data structure called *SM Page Directory* (SMPD) that stores 1 bit of information for every possible page in the virtual address space of the profiled application. Each time SMD detects that a physical page has been referenced, it sets the corresponding bit in SMPD. The page replacement algorithm must be modified to use the bit from SMPD instead of the *referenced* bit from the page table for the application under profile.

The x86 uses 4KB pages. That means there are 1 million possible pages and SMPD requires 128KB to store 1 bit for each page. We felt that this was acceptable on our prototype system. In a production kernel, memory for SMPD can be allocated on-demand.

A dedicated, 32KB write back cache was used to buffer profile writes. Since disk writes are also buffered in hardware, this dedicated cache is used to reduce the number of required communications to the I/O interface.

Linux memory management data structures keep pointers to pages that have been allocated. The vast majority of Linux applications use less than 128MB of memory, or 32K pages. We estimate that the amortized cost of sweeping through one page is no more than 30 instructions. Sweeping through 32K pages would then take approximately 1 million instructions. We did all of our profiling at 4 sweeps per second, in which case no more than 2% of the CPU will be committed to SMD. The actual observed CPU utilization was much less.

2.2 Page Replacement Policy Multiplexing

Page replacement algorithms are implemented in Linux loadable modules. When a module is loaded, it registers itself in *module table*. The system resource file serves as a registry for specifying mappings from application names to desired page replacement algorithms. Upon creation of a process, the registry table (*SM Reg-Table*) is used together with *module table* to place the correct algorithm selection in *task_struct*, a Linux data structure unique to each process.

All "get free page" requests go through *SM manager*. These requests can come from the kernel's background swapout daemon (*kswapd*), or the *page alloc* unit when requested by an application. To serve free page requests, *SM manager* round robins through all living processes and requests a page from each one (the round robin scheme actually gives consideration to process priority and attempts to leave high priority process with more memory). ***Each process uses its own choice of algorithm to choose a page to surrender***. *SM manager* uses information from *task_struct* and *module table* to make the algorithm multiplexing decision.

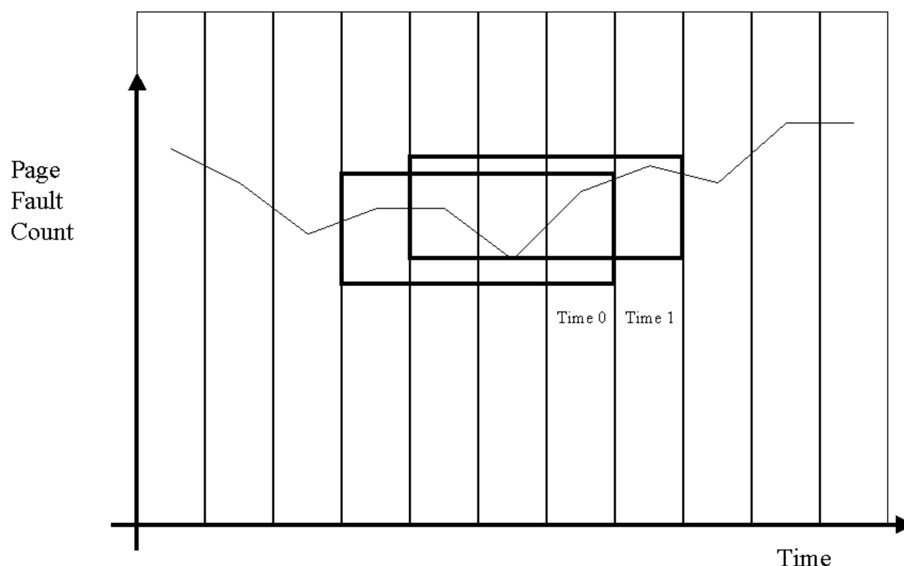


Figure 1. The vertical axis is the page fault count during a slice of time. The length of this slice is the sampling interval used by SMDD. The page fault rate at Time 0 is the average of page fault count enclosed in the window.

If an algorithm wishes to support profiling, a mechanism is exported from the kernel that allows the module to use SMPD to make replacement decisions for the profiled application.

2.2 Portability

SM should be trivially portable to Linux on all other supported platforms. 99% of the code resides in the machine independent portion of the Linux kernel. For any architecture that has at least 1 unused bit in the page table entry, the porting effort will be truly minimal.

Although SM was not designed with an eye toward other operating systems, we argue that the techniques used in the implementation of SM should be compatible with most modern operating systems. The effort required for implementation on another operating system should not be significantly greater than that of ours, which is on the scale of a class project.

3 Dynamic Intelligent Algorithm Selection

Algorithm multiplexing localizes the effect of an algorithm to one process, Dynamic Intelligent Algorithm Selection (DIAS) localizes its effect to an interval of time. DIAS dynamically adjusts page replacement algorithm selection based on page fault history information.

3.1 Algorithm

In its present incarnation, DIAS uses a first-derivative heuristic on the page fault rate to categorize page reference patterns. For each pattern, it alternates between LRU and MRU until it is satisfied that the better algorithm has been determined.

3.1.1 Page Fault Rate

Conceptually, DIAS collects the number of page faults that occurred within a regular sampling interval and keeps a history of these values. The page fault rate (PFR) is defined as the sum of page fault counts in the history buffer divided by a time value that corresponds to the length of the buffer. Thus, the PFR used by DIAS is taken over a fixed size sliding window. The sliding window moves with a resolution that is equal to the sampling interval. Figure 1. illustrates DIAS' definition of page fault rate.

3.1.2 Pattern Change Detection

DIAS also maintains a history buffer of page fault rates (not to be confused with the history buffer of page fault counts). First derivative calculations within this history buffer are used to detect a reference pattern change. The history buffer is divided into a number of segments. DIAS declares that an reference pattern has changed if all of the following conditions are met:

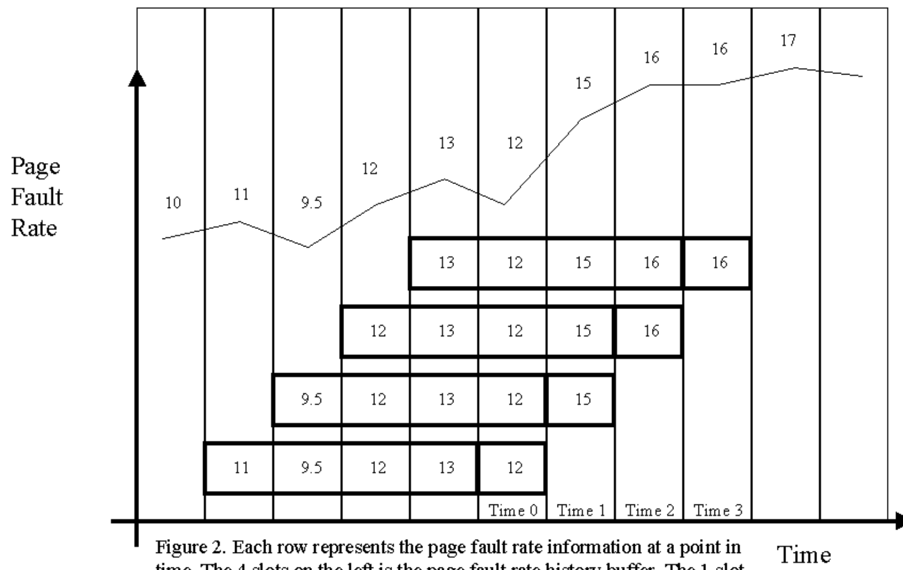


Figure 2. Each row represents the page fault rate information at a point in time. The 4 slots on the left is the page fault rate history buffer. The 1 slot on the right is the current page fault rate. Suppose that we divide the buffer into 2 segments, with proper threshold parameters, the access pattern change event will be correctly generated at Time 3.

1. For each segment of page fault rate values in the history buffer, an average difference between each of the PFR values and the current PFR is calculated. These average differences must be monotonically increasing or monotonically decreasing with respect to the age of the segment.
2. Divide the absolute value of the average difference for the earliest segment in the history buffer by the current PFR. This quotient must be greater than some fraction.
3. Divide the absolute value of the average difference for the latest segment in the history buffer by the current PFR. This quotient must be smaller than some fraction.

The above heuristic is derived from a careful study of our profile data. Our data suggests that a change in reference pattern is accompanied by a detectable change in page fault rate. The sliding window definition of page fault rate masks out errant page fault count samples.

To help capture the trend of PFR change, we divide the history buffer into segments and average their differences with the current PFR. This is desirable because the difference between adjacent sliding windows is affected by only two page fault samples, which can be taken at arbitrarily small intervals. Thus, taking the average difference helps prevent errant PFR values from blurring our vision of the longer term trend.

Condition 2 helps ensure that we are seeing a significant change in PFR, and condition 3 ensures that the PFR has stabilized to a new value.

To illustrate the algorithm, consider the situation in figure 2. The history buffer is divided into two segments. Suppose that the fraction chosen for condition 2 is 0.20, for condition 3 it is 0.05. At time 0, the average difference for segment 0 is [(11-12) + (9.5-12)]/2 = -1.75. The average difference as a fraction of the current page fault rate is abs(-1.75) / 12 = 0.145. The remaining values are tabulated below.

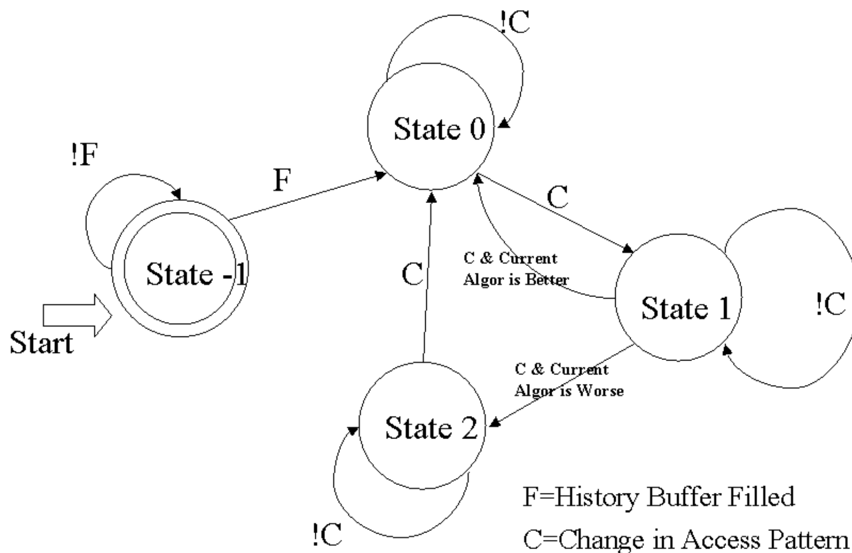
time	AD Seg 0	AD Seg 1	Fract 0	Fract 1
0	-1.75	0.5	0.145	0.042
1	-4.25	-2.5	0.283	0.167
2	-3.5	-2.5	0.219	0.156
3	-3.5	-0.5	0.219	0.031

At time 0, condition 2 fails. Condition 3 is not met at time 1 and time 2. At time 3, all conditions are met and DIAS declares that the reference pattern has changed. This is an intuitively good guess. Note that the monotonicity test did not come into play because there are only two segments.

3.1.3 Page Replacement Algorithm Selection

When DIAS detects a change in access pattern, the action taken depends on one of three possible states the process is in:

```
state 0:
      save current PFR
```



DIAS Algorithm State Transition Diagram

```
        choose the opposite algorithm
        goto state 1
state 1:
        if (current PFR >
              saved PFR from above)
              choose    the    opposite
              algorithm goto state 2
        else
              goto state 0
state 2:
        goto state 0
```

State 0 is the steady state. When DIAS detects a reference pattern change, it tries a different algorithm to see if it's better. In State 1, we are responding to the expected change in reference pattern as a result of the action taken in state 0. If the current algorithm is better, we return to the steady state. If we actually picked a worse algorithm, restore the previous algorithm and goto state 2. In state 2, we are simply waiting for the return to steady state.

When a process is first started, there is no PFR history information. Each process is started in state -1. When the history buffer fills up, DIAS jumps the process to state 0 and generates a reference pattern change event, triggering a fresh search for the best algorithm.

3.2 Implementation

DIAS is implemented in a loadable module for easy debugging and tuning. A new kernel thread was added (SM DIAS Daemon) to drive the page fault count sampling as well as DIAS itself. Page fault count is accumulated inside task_struct by the page fault handler.

The actual implemented algorithm is a much more efficient than the conceptual model presented above. Many of the abstractions can be collapsed into simple additions and subtractions. There are no non-trivial loops in the DIAS implementation. We were able to run the SMDD at 20 times per second with undetectable CPU utilization.

All parameters mentioned in the discussion above are specified in the resource file and can be updated for each new process.

In the next section, we provide some background information on a number of page replacement algorithms before we go into performance evaluation in section 5.

4 Page Replacement Algorithms

Page replacement policies have been under extensive study over the years. A large number of different page replacement algorithms have been proposed and many of them have been implemented in operating systems and database management systems. The page fault rate has been the most important criteria for choosing suitable page replacement algorithms. This is because page fault on most systems incurs secondary storage I/O operation, which may cost thousands or even millions of CPU cycles.

Of all the page replacement algorithms, the Optimal (OPT) page replacement algorithm [Aho 71] has been shown to have the lowest page-fault rate, and it does not suffer from Belady's

anomaly. The OPT algorithm says that we should always replace the page that will not be used for the longest period of time, which means we have to be able to predict the future. It is possible that we can predict the future for certain applications with extremely regular page access patterns. However, in general, the complexity of applications, the dynamic environment of machines (especially multiprocessing systems), and other random factors such as user interactions make the future very unpredictable. Thus, the OPT algorithm is basically only used for comparison with other practical algorithms.

Over the years, most people have come to the agreement that the Least Recently Used (LRU) algorithm is a good approximation to the OPT algorithm. A number of variants of LRU, such as Additional Reference-bit algorithm and Enhanced Second-chance algorithm, have been implemented in many different operating systems. LRU-K[O'Neil 00], which keeps track of the last K accesses in history instead of just last one access as in normal LRU for each page, has shown to be optimal under the assumption of independence reference model, given the same amount of information about past page accesses. Other algorithms based on counting mechanisms, such as Least Frequently Used (LFU) and Most Frequently Used (MFU) algorithms, have also been studied but not very widely used due to both their poor performance and large space requirement. The Most Recently Used (MRU) algorithm has been shown to perform well for a certain class of applications, such as database system with large sequential access.

In our work, we have carefully studied the default page replacement algorithm used on the Linux operating system. Linux uses a hybrid of LFU and LRU algorithms, but it behaves approximately like LRU. To simplify discussion, we will call it Linux LRU in the following paragraphs. Linux keeps a page age counter for each physical page in memory to inform the Kernel Page Swap Daemon (KSWAPD) whether a page is worth swapping out. The page age can be between 0 and 20 where 0 is the oldest and 20 is the youngest. When initially allocated, a page is given an age 3, and each time the page is referenced, its age increases by 3 to a maximum of 20. KSWAPD round robins through each virtual page of each process. If a virtual page resides in physical memory and has not been referenced since the last KSWAPD scan, it will decrease its age by 1 to a minimum of 0. Pages with 0 age are candidates for swapping, and a further test on the dirty bit in its page table entry and its page priority will decide if the page should be swapped out. By using this page replacement mechanism, Linux favors, and, is likely to keep in memory the pages that have been accessed recently and frequently during a past period of time.

The Linux LRU algorithm performs well for many general applications. However, other algorithms have their specialty area of applications. One most obvious example is MRU, which always swaps out the just referenced pages. In cases of sequential access or random access to pages which will not be referenced in the near future [Stonebraker 81], MRU performs much better than the common LRU algorithm. In our work, we have implemented an algorithm to approximate a Clock MRU algorithm where younger pages are much more likely to be replaced than the older pages. In the next section, we will show a performance comparison between Linux LRU and our MRU implementation on some benchmark programs.

Other page replacement algorithms that we have implemented as modules include:

- ?? An algorithm that approximates pure LRU, in which the age counter is reset to the youngest age when a page is referenced, instead of just increasing by some fixed number as in Linux LRU. Compared to Linux LRU, this algorithm is more sensitive to the very recent access than to the accumulative access numbers during earlier process execution.
- ?? LRU-2, in which we keep track of the last two accesses to a page instead of just one as in the normal LRU algorithm. This algorithm is more resistant to the sporadic changes in the page access patterns and could potentially be good for certain interactive applications.
- ?? Exponential Age Decay LRU, in which the age counter is decreased exponentially instead of just decreased by one as in the Linux LRU. This algorithm is more aggressive in replacing pages than the Linux LRU since the pages are aged much faster under the exponential decay scheme.
- ?? Round-Robin, in which the pages are examined and swapped out without the consideration of page age. This algorithm is not expected to perform better than most other algorithms, but it has a clear advantage of ease of implementation, since it does not require keeping page age information.

In the next section, we will show the results of performance tests for some of these algorithms running benchmark programs and real work applications.

5 Performance

In order to evaluate our system performance and make quantitative comparisons between different page replacement algorithms, we have run a number of tests with our system on several benchmark programs and real world applications. In this section, we show that our system has a very low overhead and has obtained some very encouraging performance results during our initial tests even without fine-tuning.

5.1 Overhead

The overhead of our page access and page fault profiling system is very low. As we have experienced so far, no significant changes to the system response time or throughput have resulted from the running our profiling daemon. A rough estimate suggests that the total overhead from our profiling system is less than 2% when we collect page access and page

fault data at a rate of four times per second. To achieve this low overhead for profiling, we have employed a high performance in-kernel file-cache for writing collected data to log files. Our current file-cache size is set to 32KB, and each log message of page access and page fault information is eight bytes long. Instead of writing the eight bytes a time to the log file, we buffer the messages in the file-cache and write the entire file-cache contents to the log file in one operation when the file-cache is full.

Our DIAS system management also has an extremely low overhead, and virtually no effect has been noticed on the performance of operating system and running applications. This is basically achieved by using a separate kernel thread, which only runs for less than 100 instructions every 50ms. On our Pentium 166MHz MMX machine, this translates to less than 1 out of 500 instruction overhead. Another optimization technique has been used to reduce the overhead in the intelligent page algorithm selection. To update the aggregate page fault numbers and window segment differences, instead of scan through the entire array in O(n) operations, we only apply addition and subtraction operations on a constant number of array elements each time. When a process is running under the DIAS system, an overhead does exist during the switching of algorithm, and this will be discussed further later in the section.

5.2 Performance Comparison

We tested our system with different page replacement modules on several benchmark programs and real world applications. The tests were all performed on a 166MHz Pentium MMX with 32MB RAM. The results were obtained without fine-tuning the system due to the time constraints. During the tests, all the modules that we have developed were loaded simultaneously. Only the algorithm resource file need to be modified to select new application and algorithm mappings, and no reboot was required.

Algor.	LRU	LFU_D	MRU	LRU2	PUREL RU
Total Page Faults	15081	18805	7524	17257	16817
Avg. PFR	38.11	40.84	29.35	40.25	40.70

Table 1: Performance comparison for FFT

Table.1 shows the results obtained when we tested the system on the Fast Fourier Transformation (FFT) benchmark program. This program allocates two arrays with sizes of about 16MB each. From the chart, we can see that our MRU implementation has outperformed the Linux LRU algorithm by 50% in terms of total number of page faults. This dramatic improvement is coming from some sequential page access pattern during the FFT execution. Figure 4 & 5. shows the page access and page fault tracing from our profiling system.

The next test result in Table 2. shows that when large array has been accessed repeatedly, MRU easily outperforms the Linux default LRU implementation. The program being run is a Matrix Multiplication (MM) benchmark program with matrices of size 1000X1000, using an optimized Robert's algorithm. In this test, the MFU also outperformed the Linux LRU implementation by a 34% margin in terms of total number of page faults. Figure 6 & 7. shows the comparison of the page access and page fault patterns of the two algorithms generated by our profiling system. And Figure 8 shows the comparison of the two algorithms in the page fault rates over time.

Algor.	LRU	LFU_D	MRU	PUREL RU	Round Robin
Total page faults	19323	20337	12857	20483	20757
Avg. PFR	33.42	33.86	24.82	33.27	35.63

Table 2. Matrix Multiplication performance comparison.

The two tests shown above provide a good example why we would like to have application specific page replacement management. Although in general LRU has a better performance than most other page replacement algorithms, in certain cases like the FFT and MM, which are both used very frequently in scientific computations, the MRU should be algorithm of choice.

Algorithm	LRU2	Linux LRU
Total # Faults	43259	39994
Avg. PFR	38.74	41.79
MIN PFR	0	0
MAX PFR	679	810
Std. Dev. PFR	70.4	85.18

Table 3. GNUPLOT performance comparison.

To demonstrate other algorithms also have their advantages, we show one more test result of running the popular graphics tool GNUPLOT with Linux LRU and LRU2 algorithms. Table 3. shows some very interesting results. Although LRU2 has about 8% more total page faults than the Linux LRU, but its average page fault rate and its standard deviation for the LRU2 are both smaller than that of the Linux LRU. (The

LRU2 tests took slightly longer to finish than the Linux LRU). This result suggests that for running certain application such as GNUPLOT, LRU2 might give a smoother performance over time, which could be a desirable property for interactive applications.

We have also run a number of other benchmark programs, such as Heap-Sort and Hanoi's Tower, and some other real world applications, such as Netscape Communicator, GZIP, Java Compiler, etc. However, due to time constraint, we have not been able to analyze the results in detail. We do expect more interesting results to come up when those results are analyzed in the future.

In conclusion, the above tests have shown that our implementation of different page replacement algorithms can give better performance for some applications than Linux LRU can. With our system, multiple page replacement algorithms can co-exist on a single operating system, and applications have total freedom of selecting which algorithm to use without affecting other processes on the machine. This fine-grained choice of algorithms would certainly provide a better overall performance to the entire system, if the algorithms are selected correctly. In the next sub-section, we show that we can even extend this fine-grained choice of algorithm one step further, so that within the same process, the page replacement algorithm used can be changed dynamically and intelligently during the execution.

5.3 DIAS at Work

DIAS is one of the most interesting parts of our work. However, again, due to the time constraint, we were only able to run a limited number of tests with our DIAS system, without any fine-tuning. However, the initial results have been very promising.

To take advantage of the DIAS system, we created a test program, Dynamite, by concatenating two benchmark programs and repeating for several times. The two benchmark programs we chose are FFT, which has shown to perform well with MRU algorithm, and an implementation of Sieve of Eratosthenes for calculating prime numbers, which in our previous tests shown to favor Linux LRU over MRU by a small margin.

Algor.	Linux LRU	MRU	DIAS
Total Faults	38213	42103	31300

Table 4. Dynamite performance comparison.

First, we ran the created test program, Dynamite, with Linux LRU only; then we ran the Dynamite with a loaded module MRU only; and we finally ran the Dynamite with our DIAS system selecting between Linux LRU and MRU. Table 4. shows the result of the tests. The result shows that our DIAS system has outperformed the Linux LRU by 18%, and outperformed the MRU by 25% in terms of total number of page faults.

The graphs in Figure 9,10,11. show a partial page access and page fault tracing of running the three algorithms, using our profiling system. The corresponding page fault rate over time graphs for each of the three algorithms are shown in Figure 12,13,14 The star marks on the X-axis in the DIAS graph show where the algorithm switching is possibly taking place. It's observable in the graphs that the DIAS result has roughly followed the page fault pattern of the better algorithm for any given period of time. However, this observed trend may also be just a coincidence and deceiving. In these tests, the page fault rate window size for DIAS and the sampling interval for the profiling system are different, and hence the pattern of page fault rate over time seen by DIAS can well be different from the page fault rate patterns shown in these graphs. This difference has to be carefully studied in the future to completely understand the behavior of DIAS.

One considerable overhead exists when the DIAS system switches the algorithm for a running process. Depending on the differences between the algorithms, the time it takes to completely recycle all the page ages could be long. We have not been able to study the effect of this overhead in detail. However, we believe this overhead should not be a major factor in the tests we have run, since the two algorithms that the DIAS chose between use very similar age counting scheme. But for future development, especially when DIAS has to choose among a larger and more diverse set of algorithms, this overhead has to be carefully studied and avoided as much as possible.

In conclusion, the DIAS was developed within a very short period of time, but the initial performance test has shown that programs running on DIAS could potentially achieve lower page faults than running on any single page replacement algorithm. Much work has to be done to fully understand the complication of DIAS.

6 Related Work

Over the history of operating system development, people have realized the importance of having application specific memory management. Previous works on this issue have given us much inspiration during the course of our work.

The Mach has implemented user level external pagers, which enabled application specific read-ahead and write-back. As an extension to Mach [Rashid 87], the PREMO [McNamee 90] supports user-level page replacement policies. However, the Mach has a micro-kernel structure, where our work is done on a commodity monolithic Linux kernel.

Hardy and Cheriton, in their ACPM [Harty 00] system, provided application controlled page-cache managers, and provided user level applications with information about the

amount of available physical memory. The ACPM is a very powerful system, but applications must be written specifically to take advantage of the ACPM. This is clearly a disadvantage, and not very practical for most commercial software. In our work, existing applications can be traced, analyzed, and applied with new algorithms, without any modification.

A former CS262 student, Peter Mattis, provided us with some of the source code from a previous class project, m_modules [Kimball 96]. M_modules provided a system call multiplexing mechanism for kernel modules and a memory management interface for Linux. The m_modules work is largely a simplified version of the ACPM, and also requires the applications to be specifically written. It did give us some considerable insights on where to begin. Unfortunately, the source code we received only contained the module multiplexing mechanism, which was of limited use to us.

At the global level, the Linux KSWAPD round robins through all the processes running, and at the local level, our page replacement modules decide their own victim page selection policies. This framework follows an observable trend in recent OS researches—two-level scheduling. ACPM does two-level scheduling by allocating a pool of free pages to an application controlled memory manager, and the manager decides its own memory usage policy. Exokernel [Engler 95] and SPIN [Bershad 00] both do low level scheduling in round-robin fashion, and provide the user-level entities the freedom of making their own resource allocation decisions. In Lottery Scheduling [Wald 94] and Scheduler Activation [Anderson 92], the kernel gives user process tickets and threads, and the user processes decide how to distribute those resources among sub-units. Nemsis [Leslie 00] also has a similar two-level resource management scheme, with additional admission control to provide consistency.

7 Experience

We chose to use Linux as our project development bed for the reasons discussed in the introduction. Free access to the entire kernel source code is obviously one advantage of using Linux. However, the lack of formality of Linux kernel also created many problems for us. The entire kernel has over 600,000 lines of C and assembly codes, but the amount of useful comments is nearly none. Although the Usenet has lots of discussion groups related to Linux, there is almost no formally published reference to the kernel internal structure and design issues. We have spent a considerable amount of time trying to figure out many minor details present in the kernel codes. And much of discoveries were done in a trial and error fashion, which required many reboots.

As we have experienced, the module facilities were extremely helpful. Linux modules basically have full access to all resources in the kernel, such as access to page tables and process task structures. And the communication between the kernel codes and module code are done directly without crossing the kernel and user space boundary. However, modules can be modified and reloaded into the kernel without rebooting. This has shortened our development and testing cycle tremendously.

One experience that surprised both of us is the seemingly ludicrous cooperative work model under which we completed the project. 95% of the programming was done by both of us simultaneously, sitting in front of the same machine! Conventional wisdom holds that no two programmers will ever agree on how to the write the same piece of code. Some how, we managed to complete an incredible amount of work in spite of the disagreements. In retrospect, one might argue that we succeeded *because* of the constant bickering that served to conduct our collective intellectual energies toward the problems we faced. We might even go further to claim that the whole was truly greater than the sum of its parts. Due to the difficulty of kernel level debugging, we rigorously visually verified our code before each testing. Much of our code worked on the first try and we experienced no logical errors that caused us any real difficulty. The effectiveness of our approach can be illustrated with our development of the DIAS system. The idea was only formalized after the poster session, two days before this writing. Then we had an eight-hour programming marathon and finished the coding and initial testing of DIAS with the reported performance results. Meanwhile, because of the care with which we designed every part of the system, no other part of SM was affected.

SM actually stands for Sky Marshall. It was named in honor of the goofy space warriors in Starship Troopers.

8 Future Work

Some of the page replacement algorithm implementations are not completely stable, and their correctness have not been rigorously verified. As a consequence, a portion of our collected data (not presented in this paper) is of dubious value.

Profile logs are compressible by a factor of 20. In light of this, profile data should be compressed before be written out to the file system. This will also make long term profiling more practical.

While cross talk is observed to be minimal, processes are not completely insulated from policies used by other processes. When a free page is requested of a process, it is not required to actually free a page, giving the process a chance to reduce memory resource available to others. When the system is highly utilized, the kernel should take a more aggressive stance against processes using a stingy algorithm, e.g. revoking a randomly chosen page. In the case of a buggy algorithm that does not return or can never fulfill the free page request, the kernel should be able to time out and kill the process all together.

Perhaps the area that beckons the most future work is DIAS. The results we present here are literally from a first attempt at a new algorithm operating under the first set of parameters we ever tried. If one considers the probable distance of our first try from an optimized version of DIAS, the initial performance results are incredibly encouraging.

We would really like to analyze DIAS for its full range of possibilities. The page fault rate in the graphs presented here are under a somewhat different definition than that used by DIAS. Once we better synchronize the two definitions, we can decide exactly when, where, and why DIAS decided to switch to a new algorithm. Overlapping the PFR vs. time plot with memory reference vs. time plot gives us an incredible amount of information that can be used to fine tune DIAS. Further, the temporal localization enabled by DIAS invites the design of new replacement algorithms that can be added to its arsenal.

Once our kernel modifications and analysis tools have been satisfactorily verified for correctness, the next challenge is to rigorously examine common uses of the operating system. Our work is only worthwhile if it improves things people care about.

Finally, we see no reason why SM and DIAS cannot be extended to other VM services, such as page allocation and prefetching. Algorithm multiplexing and dynamic selection may be *the* correct approach to implementing virtual memory services.

9 Conclusion

The same virtual memory management policy does not have to be uniformly applied to all applications at all times. It is well known that different VM policies can be beneficially applied to different memory reference patterns. The challenge lies in correctly selecting an application specific policy and ensuring that customizing toward one access pattern does not reduce system performance in the aggregate.

We presented SM, an application specific memory management system for Linux. We demonstrated that applications can individually benefit from an application-specific algorithm selection without affecting performance elsewhere. Further, we showed that dynamically selecting between two algorithms can be better than using either algorithm alone.

We believe DIAS can be dramatically improved with extensive reference pattern analysis using reliable profiling and analysis tools. To maximize the potential of these ideas, we need to demonstrate that SM can benefit the most common uses of the operating system.

REFERENCES

[Aho 71] Alfred V. Aho, Peter J. Denning and Jefferey D. Ullman
Principles of Optimal Page Replacement ACM, v. 18, no. 1, pp. 80-93, 1971

[Anderson 92] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska and Henry M. Levy
Scheduler Activation; Effective Kernel Support for the User-Level Management of Parallelism ACM Transactions on Computer System, 10(1), February 1992.

[Bershad 00] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers.
Extensibility, Safety and Performance in the SPIN Operating System.
In Proceedings of the Fifteenth Symposium on Operating Systems Principles.

[Engler 95] D. Engler, M. F. Kaashoek, J. O'Toole Jr.
Exokernel: An Operating System Architecture for Application-Level Resource Management.
ACM 1995.

[Harty 00] Kieran Harty, David R. Cheriton.
Application-Controlled Physical Memory using External Page-Cache Management

[Kimball 96] Spencer Kimball, Peter Mattis, Tracy Scott
m-Modules
CS262 Project Reports

[Leslie 00] Ian Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden
The Design and Implementation of an Operating System to Support Distributed Multimedia Applications

[McNamee 90] Dylan McNamee and Katherine Armstrong
Extending the Mach External Pager Interface to Allow User-Level Page Replacement Policies
Technical Report 90-09-05, University of Washington, September 1990.

[O’Neil 00] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum
An Optimality Proof of the LRU-K Page Replacement Algorithm

[Rashid 87] Richard Rashid, Avadis Tevanian, Michael Young, David Golub, Robert Baron, David Black, William Bolosky, and Jonathan Chew
Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures
ACM 2nd Symposium on Architecture Support for Programming Languages and Operating Systems, October, 1987

[Stonebraker 81] Michael Stonebraker
Operating System Support for Database Management
Communications of the ACM, 24(7):412-418, July 1981

[Wald 94] Carl A. Waldspurger and William E. Weihl
Lottery Scheduling: Flexible Proportional-Share Resource Management
In Proceedings of the First Symposium on Operating System Design and Implementation, Nov 1994

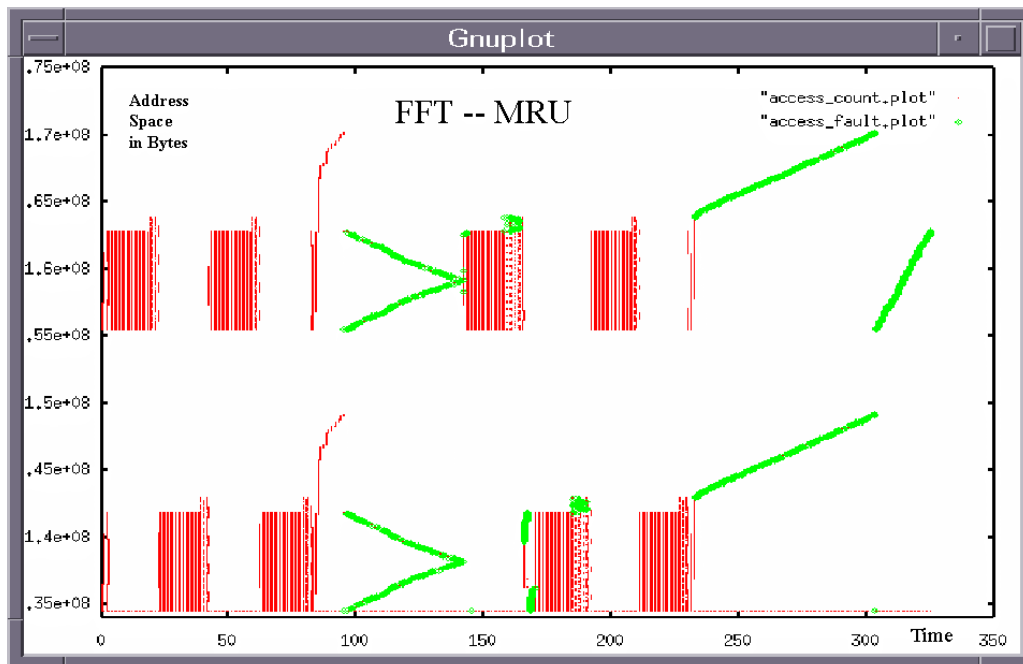


Figure 5. FFT running MRU

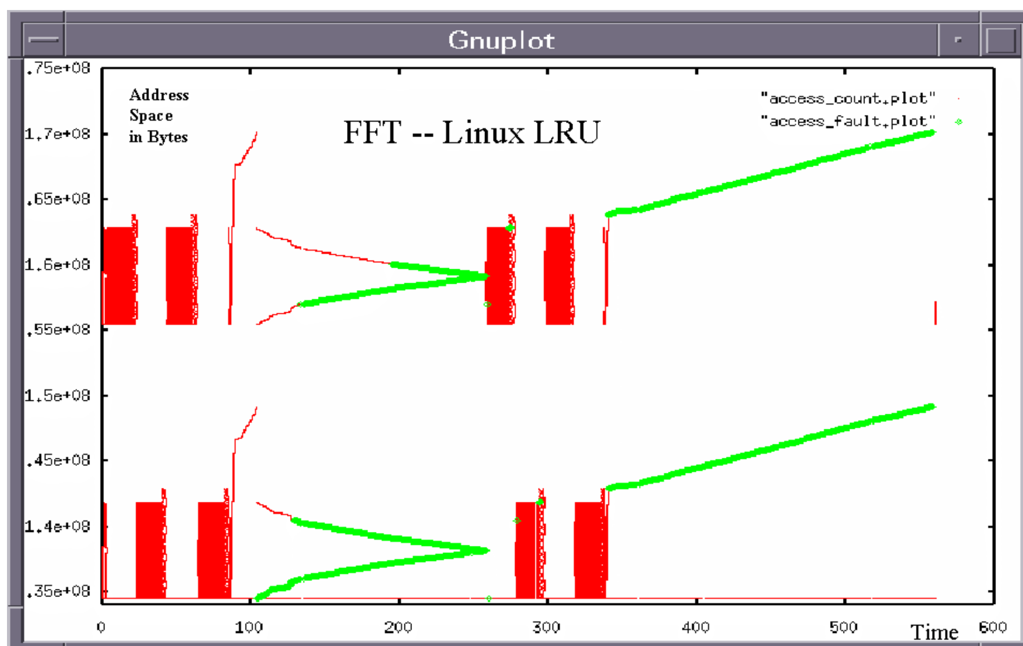


Figure 4. FFT running Linux LRU

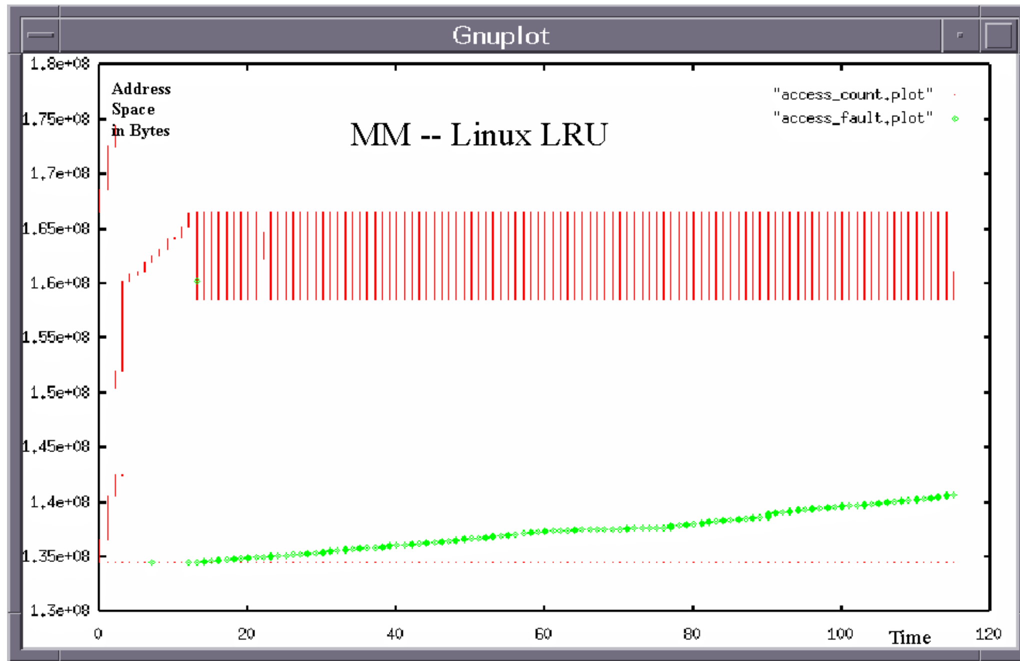


Figure 6. Matrix Multiplication running Linux LRU

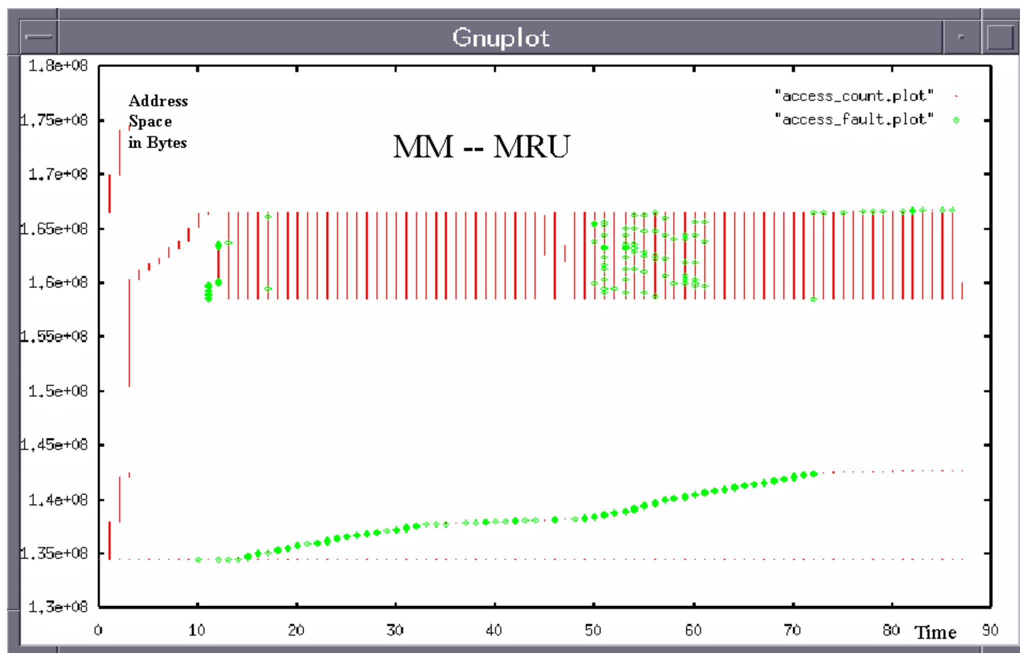


Figure 7. Matrix Multiplication running MRU

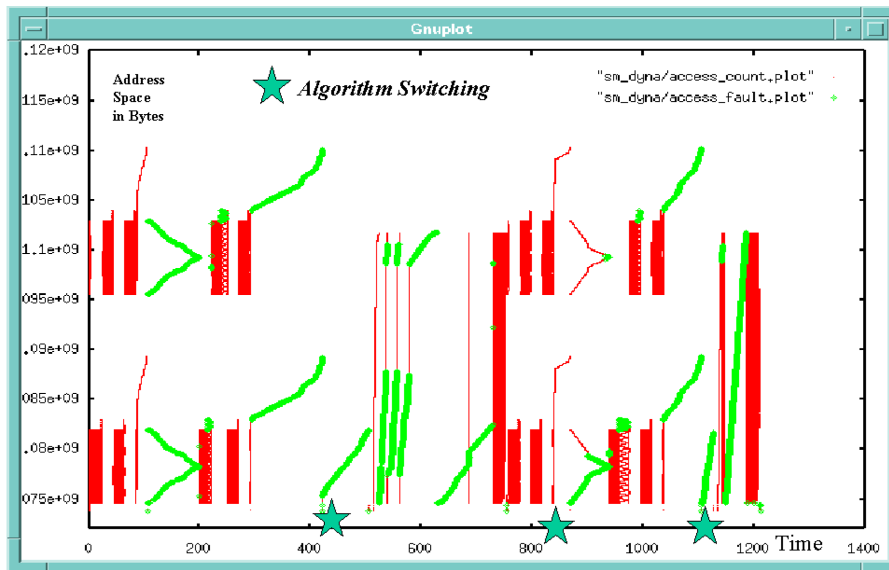


Figure 9. Dynamite running DIAS

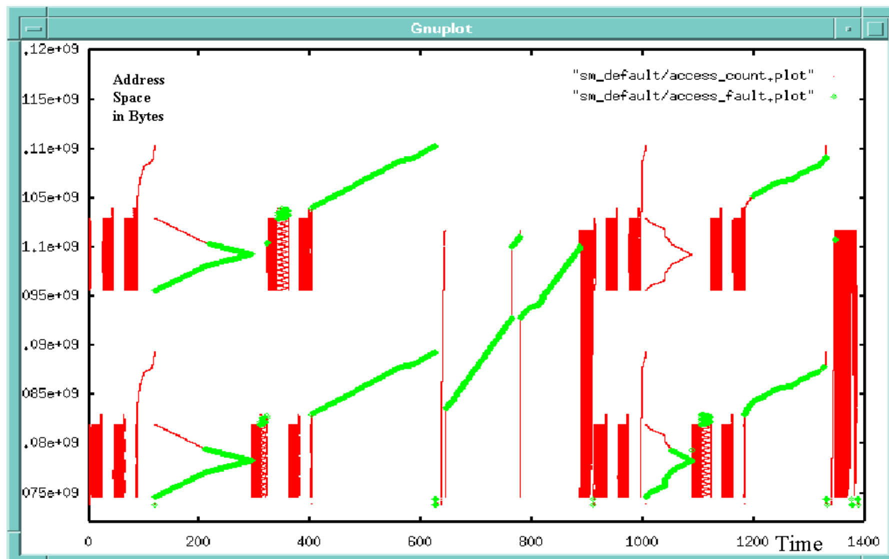


Figure 10. Dynamite running Linux LRU algorithm

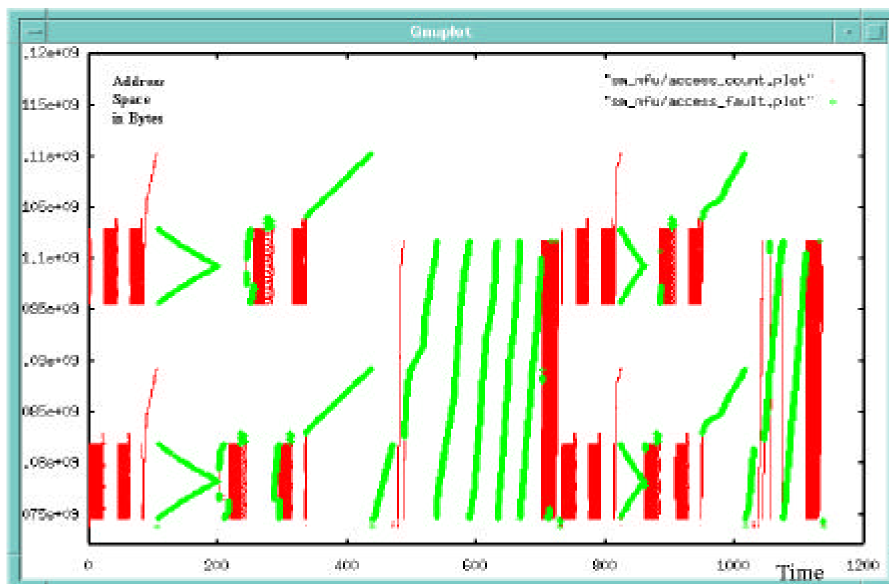


Figure 11. Dynamite running MRU

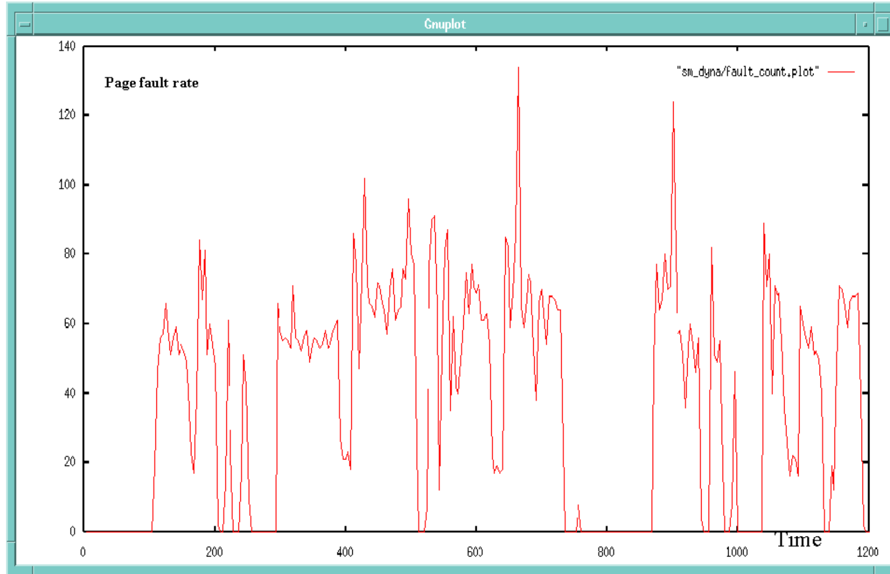


Figure 12. Dynamite running DIAS page fault rate.

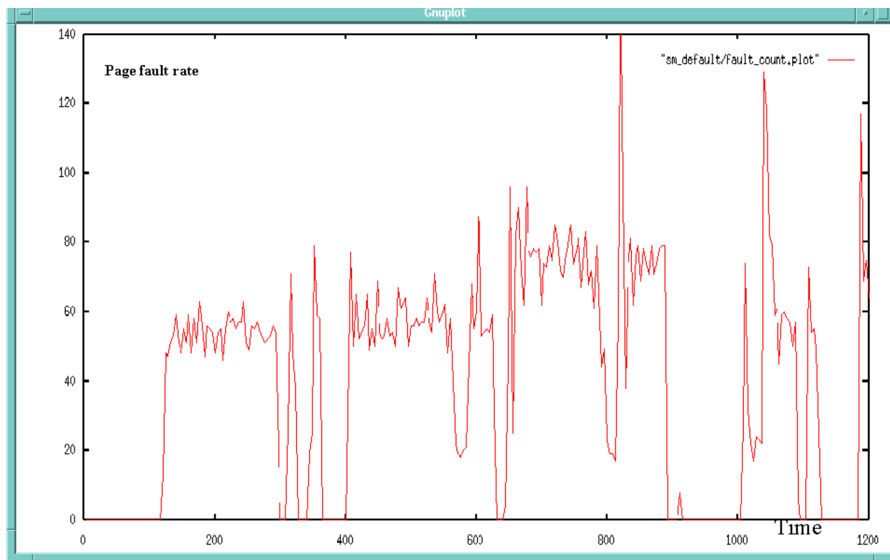


Figure 13. Dynamite running Linux LRU page fault rate.

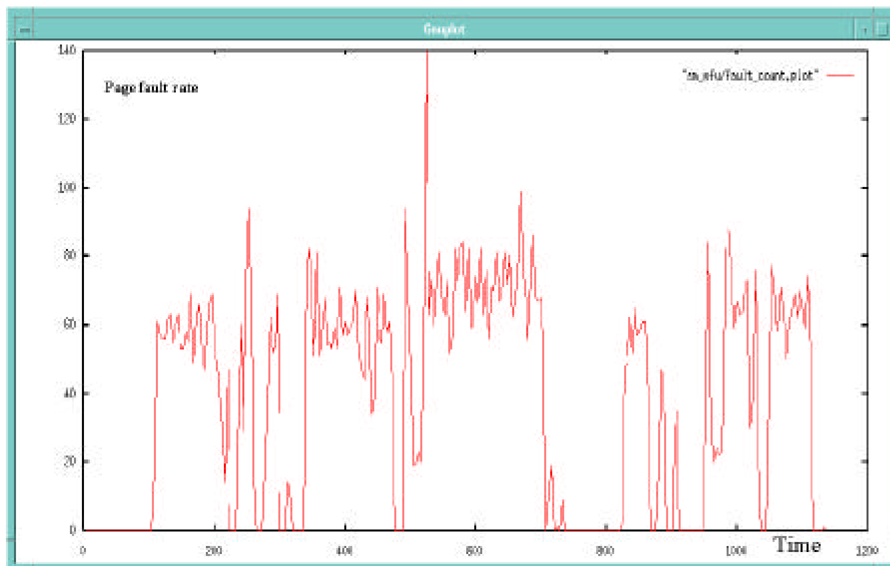


Figure 14. Dynamite running MRU page fault rate.

Appendix. Sample SM Resource File

```
# comments
# first and only first line is logging
# Field 1. kswapd frequency (HZ)
# Field 2. smd frequency
# Field 3. name of executable to log
# Field 4. logging output file
# Field 5. whether or not profiling is
done at regular intervals
4     4     dynamite    /var/log/sm.log
      true

# start application algorithm mappings
# Field 1. name of executable
# Field 2. page replacement algorithm
name
#         use one of following string --
id will be determined dynamically
#     sm_default        ----linux
default (LFU)
#     sm_rr             ----Round Robin
#     sm_lfu_d          ----Exponential
LFU decay.
#     sm_lru2     ----LRU-2
#     sm_purelru        ----Pure LRU
#     sm_mfu      ----MFU
#     sm_dyna           ----Dynamic,
self tuning
#           - Field 1 slice length (ms)
#           - Field 2 window length (#
of slices, power of 2)
#           - Field 3 number of window
segments, power of 2
#           --- pattern change
detection:
#           - Field 4 maximum percent
difference between latest segment and
#             current value (integer)
#           - Field 5 minimum percent
difference between earliest segment
#             and current value
(integer)
# Both field should not exceed 15
characters
# Field 1 can also be "all", which will
match any process unless there is
#  a special entry for a particular
process.
# all   sm_lru2
java  sm_lru2
dynamite    sm_dyna     50    16    4
      10    30
test1 sm_mfu
test2 sm_purelru
netscape    sm_lfu_d
gzip  sm_default
```